Universidade de Aveiro
2012
Departamento de Eletrónica, Telecomunicações e Informática.

Diogo Jorge Rolo Figueiral

Arquitetura Dinâmica de Controlo de Acesso

Universidade de Aveiro
2012
Departamento de Eletrónica, Telecomunicações e Informática.

Diogo Jorge Rolo Figueiral

Arquitetura Dinâmica de Controlo de Acesso

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Mestre Óscar Narciso Mortágua Pereira, Assistente Convidado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Professor Doutor Rui Luís Andrade Aguiar, Professor Associado com Agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho aos meus pais, irmão e avós pelo seu incansável apoio.

o júri

presidente

Professor Doutor José Luís Guimarães Oliveira
Professor Associado da Universidade de Aveiro

Professora Doutora Maribel Yasmina Campos Alves Santos
Professora Auxiliar com Agregação do Departamento de Sistemas de Informação da Escola de Engenharia da Universidade do Minho (Arguente Principal)

Mestre Óscar Narciso Mortágua Pereira
Assistente Convidado da Universidade de Aveiro (Orientador)

Professor Doutor Rui Luís Andrade Aguiar
Professor Associado com Agregação da Universidade de Aveiro (Co-Orientador)

agradecimentos

É com muito gosto que aproveito esta oportunidade para agradecer às pessoas que me apoiaram, tornando possível o desenvolvimento deste trabalho.
Ao professor Óscar Mortágua Pereira, meu orientador, pelo seu incansável apoio e pelo esclarecimento das dúvidas que me foram surgindo ao longo do projeto.
Ao professor Rui Luís Aguiar, meu co-orientador, pela forma como me apoiou nalgumas fases do projeto.
Agradeço também aos meus pais e avós que nunca deixaram de acreditar em mim e cujo incentivo me permitiu a realização deste curso.
A todos um “muito obrigado”.

palavras-chave

base de dados relacionais, políticas, controlo de acesso, sistemas adaptáveis, camadas de negócio.

resumo

Os programadores de aplicações de base de dados relacionais utilizam soluções de software, tais como, Hibernate, Java Database Connectivity e ADO.net, para facilitar o seu desenvolvimento. Estas soluções foram desenvolvidas com o objetivo de integrar o paradigma das bases de dados relacionais com o paradigma das linguagens de programação orientadas aos objetos. O controlo de acesso não foi contemplado por estas soluções o que conduz à necessidade de se desenvolver para cada aplicação os mecanismos de segurança considerados como necessários. Adicionalmente, em situações onde as políticas de controlo de acesso evoluem dinamicamente ao longo do tempo, não há qualquer possibilidade de ajustar automaticamente os respetivos mecanismos de controlo de acesso. Para a resolução deste problema propõe-se o desenvolvimento de uma arquitetura dinâmica de controlo de acesso para sistemas de bases de dados relacionais. Nesta arquitetura o componente principal é a própria lógica de negócio, esta contém objetos derivados de política de controlo de acesso e que são necessários para interação dos utilizadores do sistema com a base de dados. Este componente adapta-se dinamicamente em tempo de execução às alterações realizadas nas políticas de controlo de acesso que estão armazenadas num servidor de base de dados. Neste contexto, apesar de o princípio de sistemas dinamicamente adaptáveis a políticas de controlo de acesso não ser novo, as soluções de *software* existentes são algo limitadas neste aspeto. Assim, neste trabalho vamos apresentar e implementar uma arquitetura dinâmica de controlo de acesso (ADCA) que permite uma adaptação automática, em tempo de execução, dos mecanismos de controlo de acesso implementados ao nível das lógicas de negócio.

keywords relational databases, policies, access control, adaptive systems, business tiers.

abstract Developers of database applications use software solutions like Hibernate, Java Database Connectivity and ADO.net to ease its development. These solutions were developed with the aim of integrating the paradigm of relational databases with the paradigm of object-oriented programming languages. Access control was not contemplated by these solutions, leading to the need of developing for each application, the necessary security mechanisms. Additionally, in situations where access control policies dynamically evolve, there is no way to automatically adjust the respective access control mechanisms. To solve this problem we propose the development of a Dynamic Access Control architecture for relational databases systems. In this architecture, the main component is its own business logic that contains objects derived from access control policies that are required for user interaction with databases. This component adapts dynamically at runtime to changes made in the access control policies stored in a database. In this context, although the principle of dynamically adaptable systems to access control policies is not new, existing software solutions are somewhat limited in this aspect. So in this work, we present and implement a dynamic access control architecture (DACA) which allows the automatic adaptation at runtime of the access control mechanisms implemented at the business logic level.

Índice

Índice de Figuras

Índice de Tabelas

Lista de Acrónimos

ACLS	Access Control Lists
ADCA	Arquitetura Dinâmica de Controlo de Acesso
API	Application Programming Interface
CCAD	Componente de Controlo de Acesso Dinâmico
CRUD	Create, Read, Update, Delete
DAC	Discretionary Access Control
ELM	Estrutura Lógica de Memória
GUI	Graphical User Interface
IP	Internet Protocol
IUD	Insert, Update, Delete
JAR	Java Archive
JDBC	Java Database Connectivity
JVM	Java Virtual Machine
MAC	Mandatory Access Control
PCA	Políticas de Controlo de Acesso
PDP	Policy Decision Points
PEP	Policy Enforcement Points
RBAC	Role Based Access Control
SGBD	Sistemas de Gestão de Base de Dados

SGBD-R	Sistemas de Gestão de Base de Dados Relacionais
SQL	Structured Query Language
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 Introdução

Todas as áreas de desenvolvimento humano (ciência, cultura, desporto, economia, etc…) dependem cada vez mais de sistemas de informação [Sumathi, '07]. Com a evolução da tecnologia, foi possível o armazenamento de informação em sistemas computacionais, sendo a forma corrente mais utilizada para armazenamento dos mesmos, as bases de dados. Nestas são armazenados conjuntos de informações de uma forma estruturada, sendo a sua gestão realizada através de SGBD (sistemas de gestão de base de dados). Este armazenamento informatizado dos dados levou naturalmente à existência de preocupações de segurança. Entre estas preocupações, um dos aspetos críticos é formado pelas políticas de controlo de acesso (PCA), que primam pela prevenção de acesso não autorizado a informação sensível [Pereira, '12a]. Os mecanismos existentes para aplicação de políticas a dados armazenados em SGBDs, consistem na concepção de uma camada de segurança separada dos dados. Esta separação é problemática aquando da utilização de soluções de *software* (Hibernate[JBOSS, '12], JDBC (Java *Database Connectivity*)[Oracle, '12i], LINQ (Language-Integrated Query)[Meijer, '06], ADO.net[Microsoft, '12l]) que permitem obter conectividade entre linguagens de programação e uma ampla variedade de base de dados. Como estas soluções de *software* foram desenvolvidas com o foco na compatibilidade com os diferentes sistemas de gestão de dados em detrimento da aplicação de PCA[Pereira, '12a; Samarati, '01], não existe maneira de traduzir automaticamente as PCA nos mecanismos de controlo de acesso destas aplicações. Deste modo, os utilizadores podem utilizar qualquer expressão sql e executar qualquer método destas soluções de software. Este facto leva à ocorrência de erros nas tentativas de acesso a informação sensível para a qual não existe permissão. Com base neste problema, propõe-se uma nova arquitetura, ADCA (Arquitetura Dinâmica de Controlo de Acesso) que deve permitir uma fácil interação com dados existentes numa base de dados, ao mesmo tempo que se aplicam políticas de controlo de acesso. Os mecanismos de controlo de acesso definidos pela ADCA são implementados ao nível da lógica de negócio e apenas disponibilizam os acessos previamente autorizados.

1.1 Motivação

O meu interesse no desenvolvimento de aplicações de base de dados relacionais, levou à utilização de APIs como JDBC ou ADO.net. Durante a minha aprendizagem de utilização deste software foi possível verificar a existência de inconsistências nas políticas que são aplicadas na base de dados e nos comandos que posso executar nas APIs. Vamos

agora apresentar um caso exemplo para demonstrar que as soluções atuais não suportam controlo de acesso. Considere a existência de uma tabela na base de dados designada por "Tabela_Exemplo", que se encontra na Figura 1 e que se procedeu à criação de um utilizador na base de dados designado por "testuser".

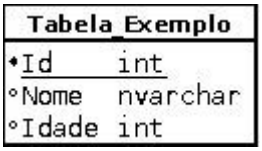


Figura 1 – Tabela Exemplo

Nesta tabela podemos verificar a existência de três campos Id, Nome e Idade. Considere que foram introduzidas as seguintes entradas representadas na Tabela 1.

Id	Nome	Idade
1	DIOGO	25
2	JOÃO	20

Tabela 1 – Dados introduzidos na Tabela Exemplo

Para este caso exemplo utilizamos a API JDBC, para a qual foi desenvolvido o seguinte código que se encontra na Figura 2.

```
stmt = con.createStatement();
rs=stmt.executeQuery("SELECT * FROM Tabela_Exemplo");

while(rs.next())
{
    System.out.println("Nome = " + rs.getString("Nome")
            + " Idade = " + rs.getInt("Idade") + "\n");

}
con.close();
```

Figura 2 – Exemplo de listagem de entradas de tabela em JDBC

Neste exemplo utilizamos a seguinte expressão SQL:

Select * From Tabela_Exemplo

Esta expressão sql permite selecionar todas as entradas da "Tabela_Exemplo". Depois da expressão sql ter sido executada através do comando *executeQuery,* é executado um ciclo while no qual são impressos no ecrã os valores dos campos Nome (getString("Nome")) e Idade (getInt("Idade")). Depois de executado o código JDBC por parte do utilizador "testuser", temos como resultado de saída:

```
Nome = DIOGO Idade = 25

Nome = JOÃO Idade = 20
```

Neste caso, como o utilizador tem acesso à tabela e a todos os campos, é apresentada no ecrã a listagem completa tal como especificada na Tabela 1. Considere agora que negamos o acesso a esta tabela através do seguinte comando sql:

```
DENY SELECT ON Tabela_Exemplo TO testuser
```

Como resultado da execução do código JDBC, obtemos uma exceção com a seguinte mensagem de erro:

The SELECT permission was denied on the object 'Tabela_Exemplo'.

Esta mensagem indica que o utilizador não possui acesso para a "Tabela_Exemplo", sendo que o programador só se apercebe desta situação depois de executar o código correspondente, nomeadamente o método *executeQuery*. Esta situação de erro ocorre também ao nível de acesso a campos para os quais não existe permissão. Para demonstrar este problema considere que se repôs o acesso à tabela "Tabela_Exemplo" e que se negou o acesso para escrita no campo Nome através do seguinte comando sql:

```
DENY UPDATE (Nome) ON Tabela_Exemplo TO testuser
```

Se tentarmos executar o código da Figura 3, em que tentamos alterar o valor do campo "Nome" para "Rui", é apresentada a seguinte mensagem de erro:

The UPDATE permission was denied on the column 'Nome' of the object 'Tabela_Exemplo'.

```
rs=stmt.executeQuery("SELECT * FROM Tabela_Exemplo");
rs.next();
rs.updateString("Nome", "RUI");
rs.updateRow();
```

Figura 3 – Exemplo de alteração de campo em JDBC

Neste caso, a exceção ocorreu na execução do método *updateRow*, o qual envia as alterações para a base de dados correspondente. Estas situações de ocorrência de erros aplicam-se também às restantes soluções desenvolvidas (ADO.net, Hibernate, LINQ e JPA). Estes problemas levaram à necessidade do desenvolvimento de uma solução que permita uma fácil interação com os dados existentes ao mesmo tempo que são aplicadas políticas de controlo de acesso. Deste modo controla-se as expressões sql que podem ser executadas por parte de um utilizador e apresentam-se apenas os métodos para os quais existe acesso. A solução passou pela construção de uma arquitetura (ADCA) que comporta diversas vertentes associadas com questões tecnológicas, sendo por isso um desafio aliciante. No desenvolvimento da ADCA são utilizados os conhecimentos adquiridos ao longo do curso e para os quais necessito de algum conhecimento extra de modo a atingir os objetivos propostos. Este trabalho apresenta situações novas para mim tais como:

- Alteração do comportamento de uma aplicação em tempo de execução;
- Construção de classes java em tempo de execução e respetivo armazenamento em ficheiros jar.

No caso da aplicação de políticas de controlo de acesso, embora já possua algumas bases, a realização deste trabalho permitiu-me adquirir um conhecimento mais aprofundado desta matéria que se encontra descrita em detalhe no ponto 2.1.

1.2 Objetivos

A ADCA tem como objetivo principal, a criação de um sistema fácil de usar e que utiliza as mais recentes tecnologias desenvolvidas (JDBC, ADO.net) para interação com SGBD's, ao mesmo tempo que são aplicadas políticas de controlo de acesso. Um esquema geral do funcionamento que queremos da nossa arquitetura encontra-se na Figura 4. Nesta figura podemos verificar o funcionamento geral da ADCA. Considere que existe uma tabela na base de dados representada a azul a qual possui informação sobre nome e idade de uma pessoa, e que pretendemos controlar o acesso a esta informação. Os utilizadores possuem um componente o qual comporta e gere a lógica de negócio. Este componente

obtém as políticas de acesso armazenadas no servidor de base de dados representado a vermelho. A lógica de negócio é construída a partir das políticas de controlo de acesso obtidas e esta aplica internamente os mecanismos de controlo de acesso, apresentado apenas os métodos que permitem aceder a informações, às quais é permitido o acesso. Neste caso podemos verificar que o utilizador 2 não possui acesso para o campo idade. Com a execução de um método da lógica de negócio, esta comunica com a base de dados onde se encontra a informação requerida (representada a azul) e obtém as informações necessárias.

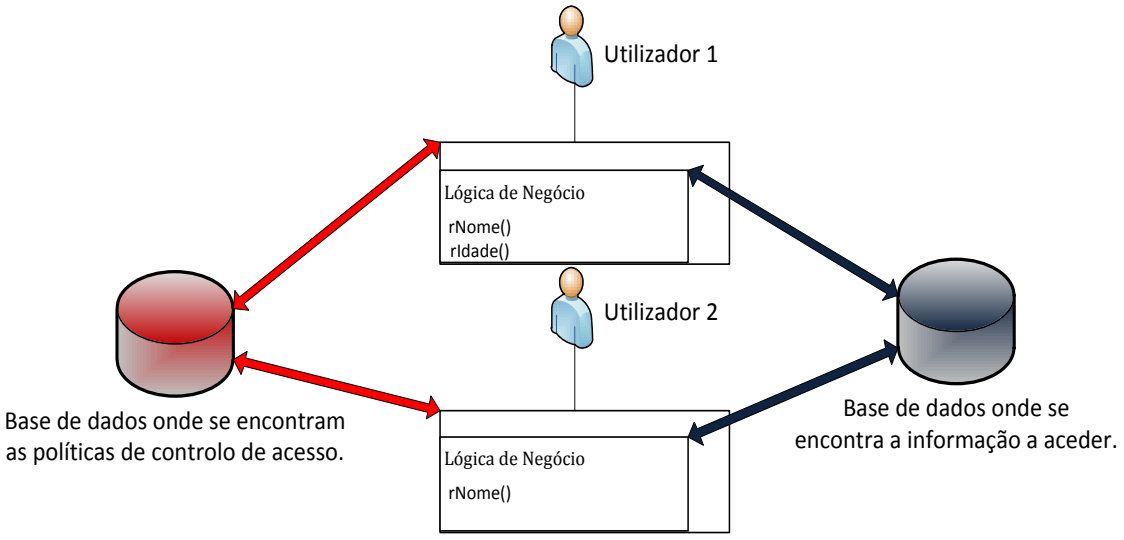


Figura 4 – Esquema geral de funcionamento da ADCA

Para a construção de uma arquitetura que permita este funcionamento é necessário:

- Definir o tipo de controlo de acesso que gere a aplicação das políticas de acesso e o local onde as políticas são armazenadas;
- Encontrar uma solução que permita a criação da lógica de negócio dinamicamente e em tempo de execução e que possua mecanismos de controlo de acesso para aplicar as políticas estabelecidas. Esta lógica de negócio deve permitir ao utilizador da ADCA aceder apenas a informação a que tenha acesso e que está armazenada numa base de dados relacional.

A lógica de negócio deverá ser desenvolvida utilizando a linguagem de programação Java, utilizando a API JDBC para a interação com a base de dados relacional.

Com a definição dos pontos necessários à criação da ADCA, é agora necessário especificar que utilizadores podem utilizar este sistema e quais as suas respetivas funções. Deste modo foram definidos três tipos de utilizadores:

- Os administradores que constroem o conjunto interfaces que compõem a lógica de negócio (esquemas de negócio) e definem as políticas de controlo;
- Os programadores de *software,* que com a disponibilização dos métodos por parte da lógica de negócio, desenvolvem aplicações relacionadas com bases de dados de uma forma facilitada;
- Os utilizadores que interagem com as aplicações desenvolvidas no ponto anterior e que são reguladas por políticas de controlo de acesso.

1.3 Organização da Dissertação

A dissertação encontra-se organizada em 5 grandes blocos:

1. Introdução: correspondente ao bloco corrente;
2. Estado de Arte e Conhecimento Prévio: neste bloco apresentam-se as várias políticas de controlo de acesso existentes e alguns trabalhos relacionados com a aplicação de controlo de acesso. Faz-se ainda uma analogia à vertente tecnológica utilizada para o desenvolvimento dos componentes que compõem a nossa arquitetura;
3. Aplicação ADCA: neste bloco apresenta-se a nossa arquitetura e também as estratégias utilizadas para o desenvolvimento dos vários componentes que a compõem;
4. Prova de Conceito: neste bloco apresentam-se as aplicações desenvolvidas que permitem a verificação do correto funcionamento da arquitetura desenvolvida;
5. Bibliografia e Referências.

2 Estado de Arte e Conhecimento Prévio

Neste capítulo, apresentamos inicialmente o que são políticas de controlo de acesso e que tipos de políticas são correntemente utilizados para controlar o acesso a dados sensíveis. De seguida descrevemos o que é um sistema de gestão de base de dados e que soluções existem atualmente para aplicação de controlo de acesso. Por fim, realizamos uma descrição detalhada das tecnologias utilizadas para o desenvolvimento da ADCA.

2.1 Políticas de Controlo de Acesso

As políticas de controlo de acesso são um passo crítico na obtenção de segurança em sistemas de tecnologia de informação. Os mais importantes requisitos de um sistema de gestão de informação são [Samarati, '01]:

- Sigilo: proteger dados e recursos contra divulgação não autorizada;
- Integridade: proteger dados e recursos de modificações não autorizadas ou impróprias;
- Disponibilidade: permitir sempre o acesso a recursos por parte de utilizadores legítimos.

A prevenção de acesso não autorizado é realizada por uma camada de segurança que nega ou permite pedidos a recursos e dados de um sistema, a este processo atribui-se o nome de controlo de acesso. Um sistema pode implementar um controlo de acesso em diferentes locais e a diferentes níveis. Por exemplo, os sistemas operativos utilizam controlo de acesso para proteção de ficheiros e diretórios, enquanto nos sistemas de gestão de base de dados (SGBD) é aplicado controlo de acesso a tabelas e vistas. O desenvolvimento de um sistema de controlo de acesso é normalmente realizado em três etapas [Samarati, '01]:

- Definição das políticas de controlo de acesso: Nesta etapa são definidas as regras (políticas) de alto nível pelo qual o controlo de acesso é regulado. Em geral as políticas de controlo de acesso são dinâmicas e evoluem de acordo com os fatores de negócio envolvidos, especificando como o acesso é gerido (a quem e em quais situações, pode aceder à informação);
- Construção dos modelos de controlo de acesso: Estes modelos permitem uma representação formal das políticas de controlo de acesso. São normalmente

desenvolvidos para mostrar as propriedades de segurança de um sistema de controlo de acesso e para mostrar as limitações teóricas desse sistema;

- Definição dos mecanismos de controlo de acesso: Neste passo realiza-se a definição das funções de baixo nível (software e hardware) que permitem que as políticas sejam impostas como descritas no modelo de controlo de acesso desenvolvido. Este componente funciona como um monitor de referência que interceta todo e qualquer acesso ao sistema e deve impor as seguintes propriedades:
    - Imune a alterações: não deve ser possível alterá-lo;
    - Intransponível: deve mediar todo e qualquer acesso ao sistema e aos seus recursos;
    - Núcleo de segurança: deve estar confinado a uma parte limitada do sistema;
    - Pequeno: deve ter um tamanho limitado para ser suscetível a métodos rigorosos de verificação.

Estas três etapas correspondem a uma separação entre as políticas e os mecanismos que as aplicam, de onde provém um grande número de vantagens. Em particular, a separação entre as políticas e os mecanismos permite uma independência entre as políticas a aplicar num lado e os mecanismos que as aplicam noutro lado. Deste modo, podemos discutir as políticas e qual o modelo a desenvolver sem nos preocuparmos com a implementação dos mecanismos para aplicação das mesmas. Um sistema diz-se "seguro", se o modelo que o compõe é seguro e se os mecanismos implementam corretamente esse modelo.

As políticas de controlo de acesso podem ser agrupadas em quatro classes principais:

- Discricionárias (DAC): políticas de controlo de acesso baseadas na identidade do solicitador e nas regras que definem o que o solicitador pode ou não realizar no sistema;
- Obrigatórias (MAC): políticas de controlo de acesso baseadas em regulações obrigatórias determinadas por uma autoridade central;
- Baseadas em papéis (RBAC): políticas de controlo de acesso que dependem dos papéis (roles) que os utilizadores têm dentro de um sistema e a cujos papéis são atribuídas regras sobre o que pode ou não ser realizado;

- Controlo de acesso baseado em credenciais: políticas de controlo de acesso utilizadas quando num ambiente aberto as entidades são anónimas entre elas.

Seguidamente apresentam-se estes tipos de políticas de controlo de acesso com mais detalhe.

2.1.1 Controlo de Acesso Discricionário

O controlo de acesso discricionário trata-se de uma política de controlo de acesso baseada na definição de um conjunto de regras de acesso (autorizações), explicitando que utilizador pode realizar uma determinada ação e em qual recurso [Capitani di Vimercati, '07]. Esta política de controlo de acesso é designada de discricionária visto que os utilizadores podem delegar os seus privilégios a outros utilizadores, enquanto a atribuição e a revogação de privilégios é regulada por uma política administrativa [Samarati, '01]. A primeira proposta por Lampson [Lampson, '74] para a proteção de recursos no contexto de sistemas operativos, foi a representação das políticas numa matriz de acesso na qual se indicam quais os utilizadores e as ações que estes podem realizar num determinado objeto. Consideremos a seguinte matriz de controlo de acesso de utilizadores a diferentes objetos do sistema operativo na Tabela 2.

Utilizador	**Ficheiro 1**	**Ficheiro 2**	**Ficheiro 3**	**Programa1**
Rui	Ler Escrever	Ler		Executar
João	Ler		Ler	
Ana	Ler Escrever			Executar

Tabela 2 - Exemplo de matriz de acesso

Nesta matriz podemos então observar que existem três utilizadores: Rui, João e Ana, aos quais são atribuídos um conjunto de ações que podem executar nos vários recursos do sistema. Mudanças ao estado do sistema são realizadas segundo a invocação de comandos que executam operações primitivas. Estes comandos permitem adicionar ou remover utilizadores e objetos, e também modificar o conjunto de ações que um utilizador pode realizar. Embora esta matriz de acesso seja facilmente implementada, esta solução é custosa visto que, em geral, a matriz é dispersa e portanto dará origem a um grande

número de células vazias. Para evitar o desperdício de recursos, a matriz de acesso pode ser implementada segundo os seguintes três mecanismos [Capitani di Vimercati, '07]:

- Tabela de Autorização: Tabela com três entradas: utilizador, ação e objeto. Cada entrada na tabela corresponde a uma autorização.

Utilizador	Acão	Objeto
Rui	Ler	Ficheiro1
Rui	Escrever	Ficheiro1
Rui	Ler	Ficheiro2
Rui	Executar	Programa1
João	Ler	Ficheiro 1
João	Ler	Ficheiro 3
Ana	Ler	Ficheiro 1
Ana	Escrever	Ficheiro 1
Ana	Executar	Programa1

Tabela 3 - Exemplo de tabela de autorização

  Podemos verificar um exemplo desta na Tabela 3, criada de acordo com os dados especificados na Tabela 2. Neste caso, a tabela não possui células em branco em oposição à Tabela 2, pois apenas são armazenadas as entradas não vazias da Tabela 2. Esta solução é utilizada no contexto de base de dados, onde as autorizações são armazenadas em Catálogos [Capitani di Vimercati, '07];
- Listas de controlo de acesso (ACLS): Cada objeto é associado a uma lista, indicando para cada utilizador as ações que esse utilizador pode exercer no objeto. Um exemplo de ACLS encontra-se na Figura 5, onde foram utilizados os dados descritos na Tabela 2 para a sua construção;
- Lista de Capacidade: Cada utilizador possui uma lista associada, designada de lista de capacidade, indicando para cada objeto, os tipos de acesso que o utilizador tem para esse objeto. Um exemplo de Lista de Capacidade encontra-se na Figura 6, na qual foram utilizados os dados descritos na Tabela 2 para a sua construção.

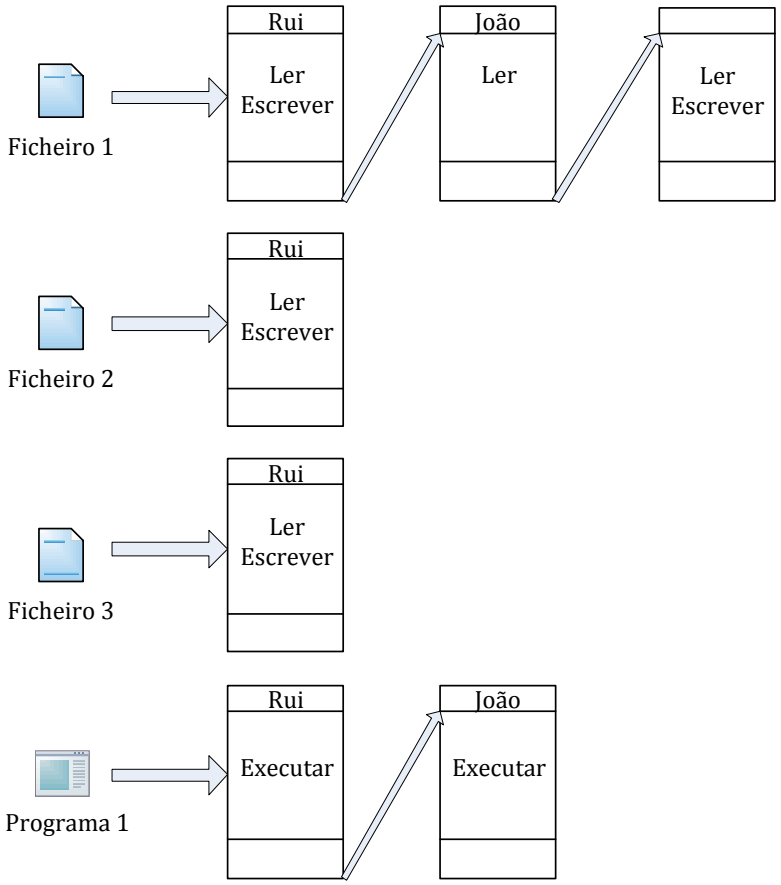


Figura 5 – Exemplo de lista de controlo de acesso

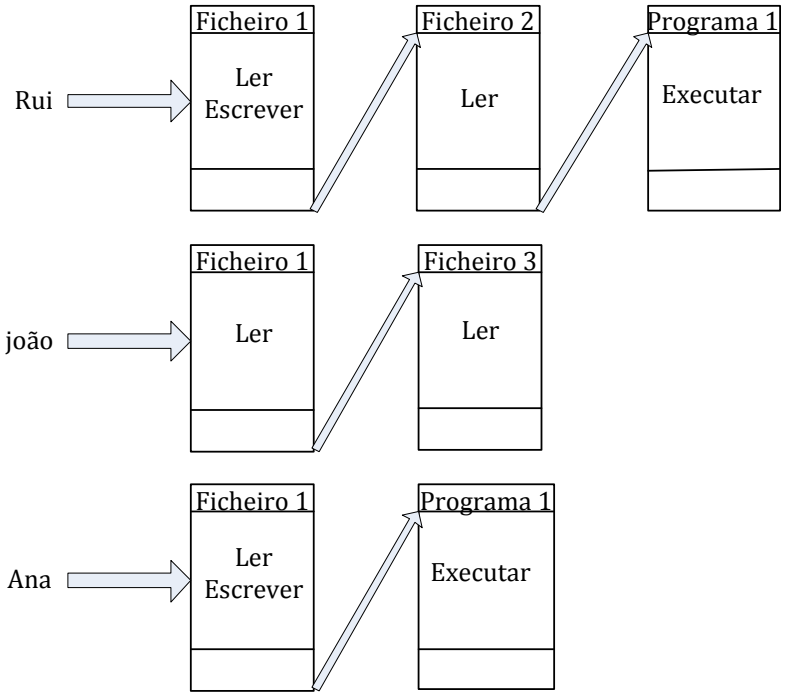


Figura 6 – Exemplo de lista de capacidades

As ACLS e as listas de capacidades apresentam as suas vantagens e desvantagens. Com a utilização de ACLS é possível verificar para cada recurso, quais as autorizações que os utilizadores têm, enquanto que para verificar todas as autorizações de um utilizador, é necessário fazer uma pesquisa a todos os objetos que a compõem. No caso da lista de capacidades é fácil verificar para cada utilizador todas as suas autorizações, mas é necessário percorrer todas as listas criadas para encontrar todas as autorizações para cada recurso. Desde que foi proposta da matriz de acesso as políticas discricionárias evoluíram bastante, permitindo também [Vimercati, '08]:

- Condições: fazem com que a validação de uma autorização dependa da satisfação de restrições específicas;
- Abstrações: para simplificação do processo de definição da autorização, utilizam-se grupos de utilizadores e classes de objetos que poderão ser organizadas hierarquicamente;
- Exceções: a definição de abstrações leva naturalmente à necessidade de suporte de exceções na definição de autorizações. Consideremos o exemplo em que todos os utilizadores, exceto o utilizador *u*, pertencentes a um grupo podem aceder a um recurso *r*. Se as exceções não fossem suportadas era necessário atribuir autorizações a todos os utilizadores, exceto u.

Vulnerabilidades e inconvenientes:

Os utilizadores deste tipo de controlo de acesso são entidades passivas, aos quais se especificam as autorizações e os quais se ligam ao sistema. Depois de ligados ao sistema, os utilizadores originam processos (sujeitos) que executam determinados pedidos ao sistema de acordo com as autorizações do utilizador. Este aspeto torna as políticas discricionárias vulneráveis a processos maliciosos, como os cavalos Tróia (Trojan Horse), os quais parecendo ser processos legítimos, contêm código escondido do utilizador que tira partido das autorizações atribuídas legitimamente.

2.1.2 Controlo de Acesso Obrigatório

As políticas de controlo de acesso obrigatório aplicam o controlo de acesso com base nas regulações de uma entidade central [Samarati, '01]. Ao contrário das políticas discricionárias, as políticas de controlo de acesso obrigatório fazem distinção entre utilizadores (utilizadores do sistema) e sujeitos (processos executados pelo utilizador). Esta distinção permite o controlo de acesso indireto (fugas ou modificações de informações), causados pela execução de processos. Estas políticas são usualmente

baseadas em classificações associadas entre os sujeitos e os objetos, a que se dá o nome de classe de acesso. Uma classe de acesso é um elemento de um conjunto parcialmente ordenado de classes. A ordem parcial é definida por uma relação dominante, a qual denotaremos por ≥. A forma mais usual de uma classe de acesso é um par de dois elementos [Samarati, '01]:

- Nível de segurança: Elemento de um conjunto hierarquicamente ordenado (Ex: Secreto, Confidencial, Não Classificado);
- Conjunto de categorias: Subconjunto de um conjunto desordenado, cujos elementos refletem áreas funcionais ou de competência (Ex: Nuclear e Exército para sistemas militares; Administração e Pesquisa para sistemas comerciais).

As políticas obrigatórias podem ser classificadas com base em sigilo ou integridade. Em seguida descreve-se em detalhe estes tipos de políticas obrigatórias.

2.1.2.1 Políticas Obrigatórias Baseadas em Sigilo

Os princípios de controlos baseados em sigilo são baseados nos modelos de segurança propostos por David Bell e Leonard La-Padula [Bell, '73] . O principal objetivo da política de controlo de acesso obrigatória baseada em sigilo é a proteção da confidencialidade da informação. O nível de segurança da classe de acesso associada com o sujeito é chamada de "clearance" e reflete o nível de confiança colocado no sujeito para não permitir acesso a informação por utilizadores que não tenham autorização. Um utilizador só deverá ter acesso à informação para a qual tem autorização. Para tal, deve ligar-se ao sistema usando a sua classe "clearance", gerando-se um processo com a mesma classe de acesso que o utilizador. O acesso é avaliado segundo estes dois princípios [Vimercati, '08]:

- Sem-Leitura-para-Cima: Um sujeito *s* pode ler um objeto *o*, só e só apenas se a classe de acesso do sujeito dominar a classe de acesso do objeto;
- Sem-Escrita-para-Baixo: Um sujeito *s* pode escrever um objeto *o*, apenas se a classe do objeto dominar a classe do sujeito.

Estes princípios previnem o fluxo de informação de sujeitos/objetos de alto nível para sujeitos/objetos de baixo nível, assegurando a proteção da informação, como se pode ver pela Figura 7.

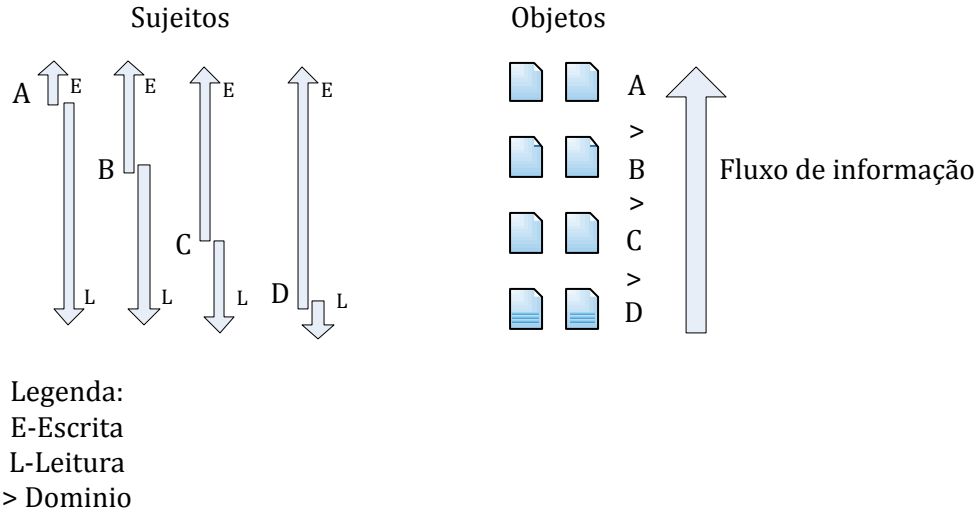


Figura 7 - Caso prático de política obrigatória baseada em sigilo (baseada na figura 6 de [Vimercati, '08])

Nesta figura podemos encontrar quatro classes de acesso (A, B, C e D), atribuídas a sujeitos e objetos. De acordo com estes princípios estabelecidos, para um sujeito *s* realizar a leitura de alguma informação de nível D (Objetos de nível de acesso D), terá que se ligar ao sistema com classe de acesso tipo D. Uma classe mais baixa na hierarquia não se traduz em menos privilégios de uma forma absoluta, mas apenas em menos privilégios de leitura. Embora estes princípios previnam um perigoso fluxo de informação, podem ser muito restritivos. Numa situação real os dados podem ter que ser rebaixados.

2.1.2.2 Políticas Obrigatórias Baseadas em Integridade

Para proteger a integridade dos dados, Biba [Biba, '77] introduziu um modelo de integridade que controla o fluxo de informação e previne sujeitos de indiretamente modificarem informação na qual não podem escrever. Tal como o modelo baseado em sigilo, neste caso, cada sujeito e objeto são associados a uma classe de integridade composta por um nível de integridade e um conjunto de categorias. O nível de integridade, neste caso, reflete a confiança colocada no sujeito para modificar ou inserir informação sensível. O acesso é avaliado segundo estes dois princípios:

- Sem-Leitura-para-Baixo: Um sujeito *s* pode ler um objeto *o*, só e apenas se a classe de acesso do objeto dominar a classe de acesso do sujeito;
- Sem-Escrita-para-Cima: Um sujeito *s* pode escrever um objeto *o*, se e apenas se a classe do sujeito dominar a classe do objeto.

Este modelo de integridade previne o fluxo de informação de objetos de baixo nível para objetos de alto nível como se pode ver pela Figura 8.

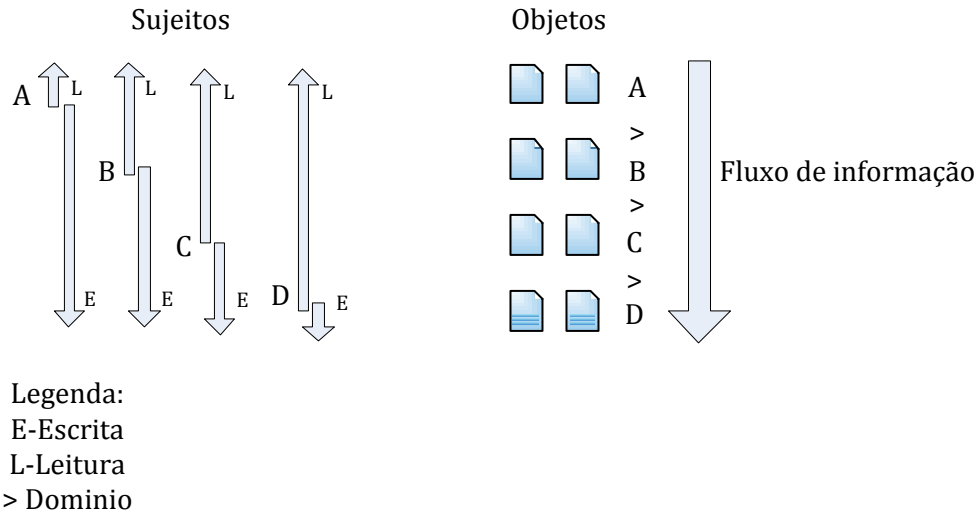


Figura 8 - Caso prático de política obrigatória baseada em integridade (baseada na figura 7 de [Samarati, '01])

Nesta figura podemos encontrar quatro classes de acesso (A, B, C e D), atribuídas a sujeitos e objetos. Neste caso, um sujeito *s* só pode realizar a escrita de informação de nível D, se a sua classe de acesso utilizada para ligação ao sistema possuir nível de integridade de tipo D. A maior limitação deste modelo é que apenas captura compromissos de integridade devido a fluxos de informação impróprios, prevenindo fluxo de informação de objetos de baixo nível para objetos de alto nível. Se pretendermos usar tanto o modelo sigilo como o modelo de integridade, é necessário utilizar duas classes de acesso, uma para controlo de sigilo e outra para controlo de integridade.

2.1.2.3 Limitações de Políticas Obrigatórias

Apesar das políticas obrigatórias protegerem melhor os dados que as políticas discricionárias, estas também possuem os seus problemas. O maior problema reside no facto de esta política de acesso controlar apenas fluxos de informação a operar de uma forma legítima, sendo então vulnerável a canais que funcionam de forma ilegítima, canais dissimulados. Estes canais não são destinados à comunicação normal mas podem ser utilizados para dedução de informações. Por exemplo, se um processo de baixo nível requerer acesso um recurso que está em uso pelo sistema por um processo de nível superior, o processo de baixo nível receberá uma reposta negativa inferindo então que

outro processo de alto nível está a utilizar o recurso. Deste modo, há troca de informação secreta e fuga de informação.

2.1.3 Políticas de Controlo de Acesso Baseadas em Papéis (Roles)

As políticas de controlo de acesso baseadas em papéis (RBAC) são uma alternativa às tradicionais discricionárias e de controlo de acesso obrigatório. O conceito de RBAC começou com a introdução de sistemas online multi-utilizador e multi-aplicação na década de 1970 [Sandhu, '96]. As políticas baseadas em papéis regulam o acesso de utilizadores conforme as suas responsabilidades organizacionais e as suas responsabilidades no sistema. Cada papel representa um conjunto de ações e atividades associados com uma entidade trabalhadora. Um papel pode designar o que um utilizador é num sistema (professor) ou pode ser mais específico e refletir uma tarefa que pode ser realizada (Inserir_Notas).

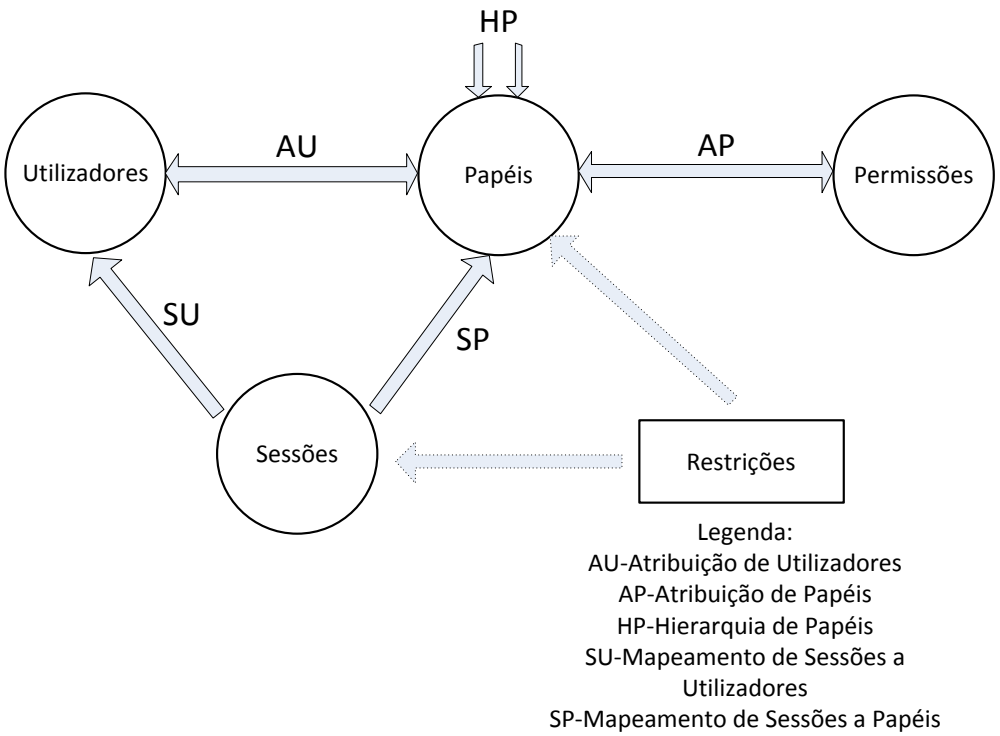


Figura 9 - Exemplo de modelo RBAC baseado no RBAC96

As autorizações de acesso são definidas nos papéis e aos utilizadores é dada autorização para os usarem. Esta representação explícita através de papéis permite simplificar os modelos de segurança em sistemas de grandes dimensões, como se ilustrará mais à frente.

Sandhu no paper [Sandhu, '96] introduziu o bem conhecido modelo RBAC96, que fornece um ponto de trabalho no qual podem ser acomodadas várias variantes do RBAC. Na Figura 9 apresenta-se um esquema básico do modelo introduzido por Sandhu. Neste caso podemos encontrar um conjunto de papéis que se agrupam de uma forma hierárquica e aos quais são atribuídas permissões. Aos utilizadores são consequentemente atribuídos papéis, enquanto as sessões guardam informações sobre utilizadores e papéis. Em seguida descrevem-se os elementos deste modelo em detalhe [Sandhu, '96]:

- Utilizadores e papéis: O utilizador é um ser humano. Papel é o nome de uma função de trabalho que descreve a autoridade e as responsabilidades conferidas a um utilizador desse papel;
- Permissões: Uma permissão é a aprovação de um modo particular de acesso a um ou mais objetos do sistema. Permissões são sempre positivas e conferem a quem as possui a habilidade de realizar ações no sistema. Este modelo conceptual acomoda muitas interpretações para permissões onde o acesso pode ir desde uma sub-rede inteira até um campo particular numa tabela particular.

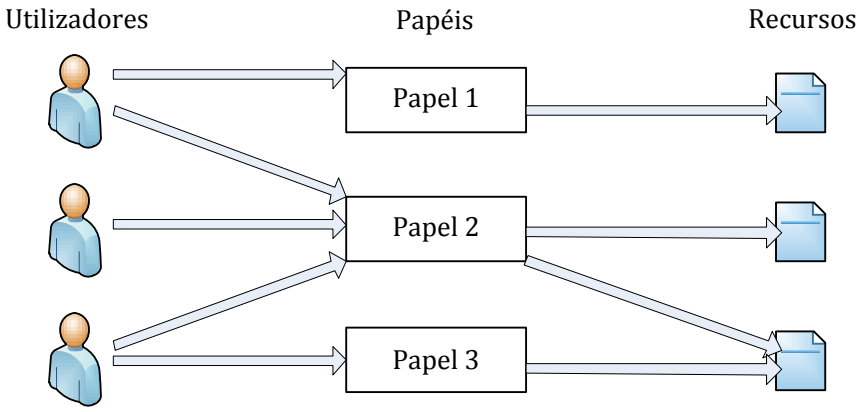


Figura 10 - Exemplo de permissões atribuídas num modelo RBAC

A Figura 10 mostra as relações de atribuição de utilizadores (AU) e atribuição de permissões (AP). Ambas são relações muitos para muitos e ambas são chaves para o controlo de acesso baseado em papéis. Um utilizador pode pertencer a vários papéis e um papel pode ter muitos utilizadores. O papel é uma posição intermédia que permite ao utilizador exercer a permissão e aumentar assim a facilidade de gestão de utilizadores e permissões, como se verá mais à frente;

- Sessões: Uma Sessão é um sistema de troca de informações semi-permanente também conhecido como diálogo, conversação ou reunião, entre dois ou mais dispositivos de comunicação. Quando os utilizadores estabelecem sessões, são lhes atribuídos um conjunto de papéis aos quais pertencem. Quando um utilizador cria uma sessão, pode selecionar quais os papéis que pretende ativos, podendo estes serem futuramente alterados consoante o critério do utilizador. Estes papéis podem ser utilizados por mais que um utilizador ao mesmo tempo (multirole) como se pode verificar pelo esquema da Figura 10, onde o Papel 2 se encontra atribuído a três utilizadores;
- Restrições: As restrições têm um aspeto fundamental no RBAC, sendo um importante mecanismo para estabelecer políticas organizacionais de alto nível. É um mecanismo pelo qual os administradores podem restringir a habilidade dos utilizadores de ter determinados privilégios. Podem ser aplicados nas relações Utilizadores-Papéis, Papéis-Permissões e no mapeamento de utilizadores e papéis a sessões. Existem os seguintes tipos de restrições:
    - Papéis mutuamente exclusivos: A um mesmo utilizador só pode ser atribuído no máximo, um dos papéis;
    - Cardinalidade: Neste tipo de restrição podemos definir o número de membros de um mesmo papel e similarmente o número de papéis a que um utilizador individual pode aceder;
    - Papéis Pré-Requeridos: O conceito de pré-requisito consiste na competência e na adequação, podemos definir que a um utilizador só pode ser atribuído o papel A se a este já tiver atribuído o papel B;
    - Outras restrições podem ser temporárias. Como por exemplo:

        a)Restringir o número de sessões por utilizador;

        b)Atribuir a restrição à própria hierarquia de papéis, em que se restringe por exemplo, a possibilidade de uma papel pai possuir as permissões de um determinado papel filho.

- Hierarquia de Papéis: Em muitas aplicações existe uma hierarquia normal de papéis em que a atribuição de uma autorização a um papel num nível hierárquico superior permite uma propagação das autorizações aos papéis de um nível inferior, como se pode ver pela Figura 11. Neste caso, se fosse atribuída a autorização de Gestor, e como as autorizações se propagam hierarquicamente,

havia uma herança das autorizações subjacentes (Gestor De Loja, Gestor De Escritório e Gestor de Vendas).

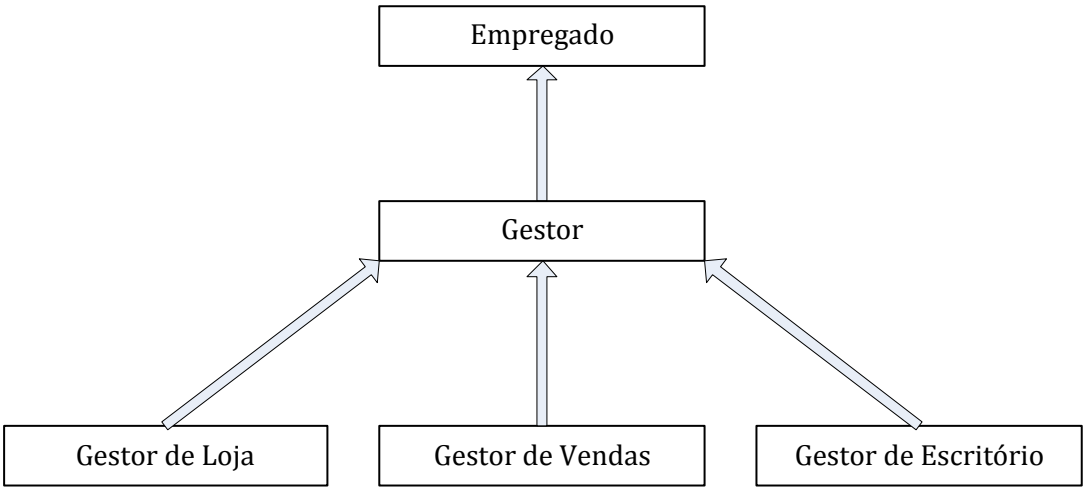


Figura 11 - Exemplo de Hierarquia de Papéis

A conjunção destes elementos permite chegarmos a uma base sólida para aplicação de RBAC. Um modelo RBAC pode nem sempre descrever todos estes elementos. Deste modo, foram estabelecidos os seguintes modelos baseados no modelo RBAC da Figura 9 [Sandhu, '96]:

- Rbac0: Modelo base, consiste em todos os elementos presentes na Figura 9 exceto, hierarquia de papéis (HP) e restrições;
- Rbac1: Neste modelo são introduzidos os elementos do rbac0 mais a inclusão de hierarquia de papéis;
- Rbac2: Neste modelo são introduzidos os elementos do rbac0 mais a inclusão de restrições;
- Rbac3: Modelo consolidado, combina os elementos dos modelos Rbac1 e Rbac2, oferecendo ao mesmo tempo hierarquia de papéis e restrições.

Vantagens da utilização de RBAC

A utilização deste tipo de controlo de acesso traz as seguintes vantagens[Samarati, '01]:

- Gestão de autorizações: Simplificação na gestão de políticas de segurança, visto que existe atribuição de papéis a utilizadores e atribuição de autorizações a papéis. Quando um utilizador entra no sistema são-lhe atribuídas permissões a papéis, podendo estas serem alteradas pelo administrador do sistema;

- Utilização de Hierarquia de Papéis;
- Menor Privilégio: Aos utilizadores só são atribuídas as permissões necessárias. Utilizadores com papéis de alto nível hierárquico não precisam de os usar até serem necessários. Isto aumenta a segurança contra erros inadvertidos ou intrusões ilegítimas;
- Separação de deveres: A separação de deveres é o conceito de ser necessário mais que uma pessoa para completar uma tarefa. Por exemplo, a pessoa que gere o escritório, não deve ser a mesma pessoa que realiza as vendas na loja. A separação de deveres pode ser aplicada quer estaticamente (definindo papéis conflituosos, isto é, papéis que não podem ser executados pelo mesmo utilizador) ou dinamicamente (ao aplicar o controlo em tempo de execução);
- Aplicação de restrições: Utilizando restrições podemos restringir a habilidade dos utilizadores de ter determinados privilégios. Por exemplo, podemos restringir a cardinalidade de um papel, ou seja restringir o número de utilizadores que o podem utilizar. Estas também podem ser dinâmicas pois pode-se restringir a utilização de um papel em determinadas circunstâncias. Por exemplo, restrições temporais em que só se pode utilizar um papel num tempo bem definido.

Delegações

Delegação de papéis em RBAC é a habilidade de um utilizador (designado de delegando), que é membro de um papel, autorizar outro utilizador a tornar-se membro desse papel [Barka, '00]. Por exemplo, considere que o gestor de escritório da Figura 11 necessita de aceder às vendas para calcular os lucros da loja. Neste caso um utilizador designado para a role Gestor De Vendas podia delegar esse papel ao utilizador Gestor de Escritório, ficando este segundo utilizador com acesso aos dois papéis. Deste modo o Gestor de Escritório pode aceder aos dados sobre as vendas realizadas. Cada papel delegado possui dois tipos de membros:

- Membros originais: Membros que são originalmente atribuídos àquele papel pelo administrador do sistema;
- Membros delegados: Membros aos quais são atribuídos os papéis delegados pelos membros originais.

Este tipo de delegações devem ter uma duração associada. Deste modo devem ser revogados os direitos anteriormente atribuídos ao fim de algum tempo. Para tal foram definidos dois tipos de revogações[Barka, '00] :

- Revogação usando time-outs: Neste caso são atribuídos tempos a cada delegação atribuída, de modo a que quando este tempo expirar, a delegação também expira;
- Revogação Humana: Neste caso apenas o membro delegante (membro original que atribuiu a delegação) pode revogar a delegação. Esta aproximação possui algumas vantagens e desvantagens. As vantagens são as seguintes [Barka, '04]:
    - O membro original pode acompanhar e controlar o comportamento do membro delegado temporário;
    - Minimiza a possibilidade de conflitos entre os membros originais que podia resultar em outro membro sem ser o membro delegante original a revogar a delegação atribuída. Esta aproximação traz uma desvantagem, que consiste na existência de comportamento errático por parte do membro delegante, o que pode levar à existência de um acesso aos recursos atribuídos pelo papel delegado por demasiado tempo, até a ocorrência de um time-out.

Delegações em hierarquia de papéis

Em [Barka, '00] é também definida a forma como são realizadas as delegações em RBAC com utilização de uma estrutura hierárquica de papéis. Numa hierarquia de papéis, os papéis pai herdam as permissões de papéis que são filhos. Temos então vários tipos de delegações que se podem utilizar no modelo rbac, umas acarretando mais riscos que outras:

- Delegações ascendentes: Neste caso, existe atribuição de delegação de um papel a um papel pai deste. Como numa estrutura hierárquica, os papéis pai herdam as permissões dos papéis filhos, esta delegação, é neste caso inútil;
- Delegações descendentes: Delegação onde existe atribuição de um papel a um papel mais baixo no nível hierárquico. A delegação descendente funciona parcialmente, visto que, não podemos delegar o todo do papel pois levaria a um encolhimento da hierarquia;
- Delegações em corte-transversal: Este tipo de delegação é muito útil pois como podemos verificar pela Figura 11, o gestor de vendas pode delegar o seu papel ao gestor de escritório, por exemplo, para este fazer uma verificação das contas.

2.1.4 Políticas de Controlo de Acesso Baseadas em Credenciais

Em ambientes abertos e dinâmicos, tais como a Internet, a decisão de conceder acesso a um recurso é geralmente baseada nas características do solicitante em vez de na sua identidade[Damiani, '05]. As partes são então desconhecidas entre si e podem desempenhar tanto o papel de cliente (que recebe os recursos) como o papel de servidor (que fornece os recursos). Soluções avançadas de controlo de acesso devem então decidir, a que cliente deve ser dado acesso e que servidor está qualificado para fornecer o mesmo recurso. As soluções iniciais utilizavam o conceito de Trust Management onde eram vinculadas chaves públicas a autorizações. Este método tinha uma grande desvantagem [Vimercati, '08]: A especificação das autorizações era difícil e a identidade do utilizador era descartada. Este problema foi resolvido com a introdução de certificados digitais que permitem identificar as autorizações e a identidade dos utilizadores.

O acesso de controlo é então realizado pela troca de mensagens entre as entidades. Esta troca é realizada da seguinte maneira [Vimercati, '08]:

- O cliente realiza um pedido para um recurso;
- O servidor analisa se o cliente forneceu as credenciais necessárias. Se o cliente forneceu todas as credenciais, o servidor concede acesso, caso contrário, envia a informação sobre as credenciais que ainda são necessárias;
- O cliente seleciona as credenciais requeridas pelo servidor, se possível, e envia-as. Se as credenciais satisfazem o pedido o servidor, este concede acesso ao recurso.

Desta maneira realiza-se o controlo de acesso em ambientes abertos, onde os elementos são desconhecidos entre eles.

2.2 Abordagens Correntes para Aplicação de Controlo de Acesso

Neste capítulo expomos inicialmente as principais características das bases de dados relacionais e que modelos já existem para aplicação de políticas de controlo de acesso. De seguida são apresentados trabalhos realizados no âmbito da aplicação de controlo de acesso.

2.2.1 Sistemas de Gestão de Base de Dados Relacionais

Um sistema de gestão de base de dados (SGBD) consiste numa coleção inter-relacionada de dados e um conjunto de programas para acesso aos mesmos [Sumathi, '07]. Os principais objetivos de um sistema de gestão de base de dados são:

- Disponibilidade dos dados;

- Integridade dos dados;
- Segurança dos dados.

Neste trabalho focamo-nos num tipo específico de SGBD, o sistema de gestão de base de dados relacional (SGBD-R). Os SGBDR são baseados no modelo relacional introduzido por E.F. Codd em [Codd, '83]. O termo relacional refere-se ao modo como o SGBDR espera que estejam organizados os dados que está a gerir. Uma relação é uma tabela de informação que está organizada em linhas e colunas. Por exemplo considere o caso, onde se pretende a criação de uma tabela para armazenamento de informação de uma Pessoa. A tabela criada pode ser a representada pela Figura 12.

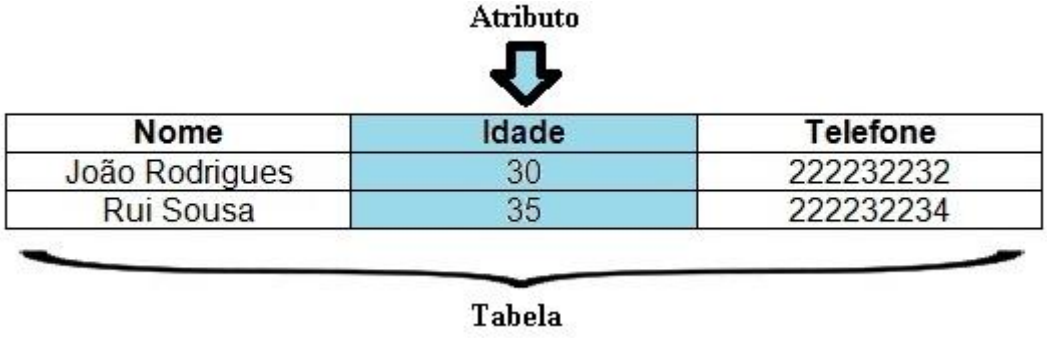


Figura 12 - Esquema de tabela de base de dados relacional

Nesta figura, podemos verificar a existência de um conjunto de linhas que contêm um conjunto de atributos (*Nome*, *Idade* e *Telefone*). Cada atributo possui o mesmo tipo de dados, podendo ter valores diferentes. No caso do atributo *Idade*, este pode ser representado pelo tipo *int* na base de dados relacional, representado um número inteiro. Para possibilitar a criação de relações entre tabelas foi utilizado o conceito de chave.

Os dois tipos principais de chaves são:

- Chave Primária: Atributo ou conjunto de atributos que possuem a propriedade de identificar de forma única uma linha da tabela;
- Chave Estrangeira: campo numa tabela que coincide com a chave primária de outra tabela.

Considere a tabela da Figura 12 e considere a existência de uma tabela "Vendas", onde se armazenam os dados relativos às vendas, que inclui uma referência para o Cliente. Neste caso temos que utilizar ou criar um atributo na tabela Cliente para funcionar como

chave primária. O atributo Nome não deve ser utilizado como chave primária por duas razões:

- Chave Primária tem que ser única: neste caso pode haver repetição de nomes (primeiro e ultimo nome);
- A utilização de dados textuais como chave primária reduz a performance obtida, sendo muito melhor utilizar números inteiros, pois permitem uma comparação mais rápida e um consequente aumento de *performance*.

Os atributos *Idade* e *Telefone* podem conter valores repetidos. Deste modo foi criada uma nova coluna (coluna *Id*) com números inteiros únicos para identificar as chaves primárias.

Id	Nome	Idade	Telefone
1	João Rodrigues	30	222232232
2	Rui Sousa	35	222232234

Tabela 4 - Tabela Cliente com inserção de atributo Id

A nova tabela Cliente está representada na Tabela 4 onde são utilizados os campos da coluna *Id* para representação da chave primária. Para a tabela Vendas referenciar a tabela Cliente é criada uma coluna com a respetiva chave estrangeira referente à chave primária da tabela Cliente.

A verificação da consistência dos dados existentes numa base de dados relacional é realizada através de restrições de integridade, sendo garantida pelo próprio SGBD. Existem entre outras, os seguintes tipos de restrições de integridade definidos por E.F.Codd[Codd, '83]:

- Integridade de Domínio: O SGDB verifica se os dados são do tipo permitido e se estes podem ter valores nulo ou não. Por exemplo, na nossa tabela Cliente, o atributo idade é do tipo inteiro e por isso qualquer introdução de um valor com tipo diferente de inteiro é negado pelo SGBD;
- Restrição de Entidade: Na declaração de um atributo como chave primária, o SGBD não permite que este tenha valores duplicados ou nulos;
- Integridade Referencial: Uma chave estrangeira de uma tabela tem que coincidir com uma chave primária da tabela a que a chave estrangeira se refere. Considere que a tabela Vendas contém uma entrada com a chave

estrangeira a referenciar a primeira entrada da tabela Cliente. Neste caso, quando se tenta apagar a primeira entrada da tabela Cliente, o SGBD não o permite, visto que está referenciada na tabela Vendas;

- Integridade definida pelo Utilizador: Restrições de integridade mais complexas definidas pelos utilizadores. Exemplo:
    - O número de telefone ter necessariamente 9 dígitos;
    - Idade não ter valores impossíveis, Idade<0 ou Idade>200.

A criação de uma base de dados relacional é precedida pela construção de modelos com as tabelas e as relações. Estes modelos permitem uma melhor organização e redução da redundância, fazendo com que construção das relações entre tabelas seja melhor estruturada. Foram definidos os seguintes modelos:

- Modelo Conceptual: Descreve de uma forma simples e de fácil compreensão, as informações de um contexto de negócio. Neste modelo devem estar identificadas as entidades e o relacionamento entre elas;
- Modelo Lógico: O modelo lógico possui mais detalhe que o modelo conceptual. Neste modelo define-se os atributos das entidades, as chaves primárias e as chaves estrangeiras;
- Modelo Físico: É construído a parir do modelo lógico e descreve as estruturas físicas do armazenamento dos dados, tendo em conta as limitações impostas pelo SGBD escolhido.

A maioria das implementações do modelo relacional utilizam a Linguagem de Consulta Estruturada (SQL) para definição dos dados e consulta dos mesmos. Embora as bases de dados baseadas em SQL desviem-se do modelo relacional em muitos detalhes, esta tornou-se numa das linguagens de base de dados mais utilizadas.

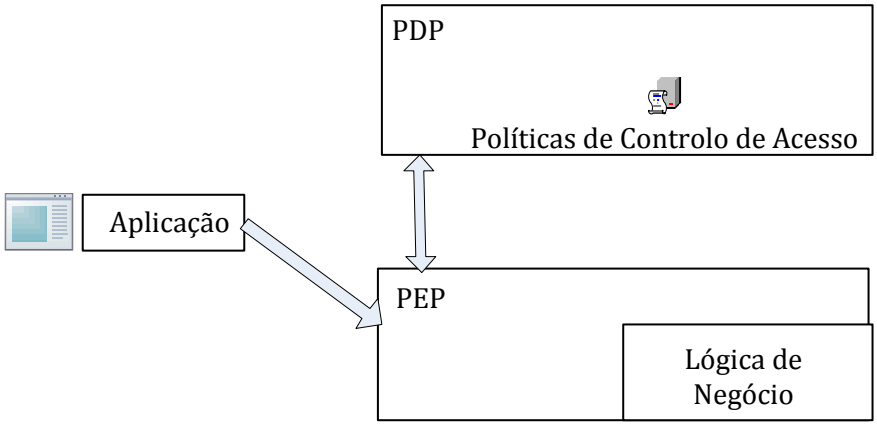
2.2.2 Mecanismos Correntes para Aplicação de Políticas de Controlo de Acesso a SGBD-R

Os mecanismos correntes para aplicação de políticas de controlo de acesso a SGBD-R consistem em criar uma camada de segurança separada, seguindo uma de duas aproximações [Pereira, '12a] tradicional e PEP-PDP:

- A aproximação tradicional [Sandhu, '94] foi implementada nas bases de dados relacionais desde os primeiros produtos. Neste caso existem três tipos de controlo de acessos aplicados:
    - Acesso de controlo discricionário: neste caso utiliza-se os privilégios atribuídos pelos *standards* da linguagem sql. O criador de uma relação numa base de dados torna-se o seu dono e tem a habilidade de atribuir a outros utilizadores o acesso a essa relação, através de comandos como *GRANT* ou *REVOKE*. Uma das desvantagens deste tipo de utilização e uma das principais fraquezas é que, mesmo que o acesso a uma relação seja estritamente controlado, é possível a um utilizador com um comando SELECT criar uma cópia da relação, contornando assim o controlo de acesso aplicado;
    - Acessos de controlo obrigatórios: são aplicados a base de dados, introduzindo classificações de segurança e "clearance" de segurança;
    - Controlos de acesso baseados em papéis: neste caso são utilizados papéis para atribuição de permissões, permitindo uma maior facilidade na gestão dos controlos de acesso, em relação ao controlo discricionário encontrado na linguagem sql standard. Estes tipos de controlo são geridos automaticamente pelo SGBDR e transparentes aos utilizadores, notando-se a sua presença apenas aquando da execução de uma instrução que vá de encontro às políticas de acesso de controlo definidas.
- A Aproximação PDP-PEP (cujo esquema encontra-se na Figura 13) e na qual existem dois componentes principais:
    - PDP (Policy Decision Point-Ponto de Decisão de Políticas): é o ponto no qual as decisões de políticas são realizadas. Aqui encontram-se as políticas de acesso e também o mecanismo que recebe os pedidos vindos da lógica de negócio (PEP) e que retorna respostas normalmente a permitir ou a negar o acesso;
    - PEP (Policy Enforcement Point-Ponto de Aplicação de Políticas): mecanismos inseridos na lógica de negócio e responsáveis pela aplicação das políticas. Antes de algum serviço ser executado, o PEP

envia um pedido ao PDP para verificar se o pode executar. Se o acesso é permitido o serviço é executado se não o PEP nega a execução do serviço.

Esta arquitetura permite assim a separação entre a localização onde se encontram as políticas de acesso e onde estas políticas são aplicadas. Se as ações são permitidas pelo PDP, o PEP executa-as na base de dados.



Legenda:
PDP-Ponto de Decisão de Políticas
PEP-Ponto de Aplicação de Políticas

Figura 13 - Exemplo de arquitetura PEP-PDP

2.2.3 Trabalhos Relacionados com a Aplicação de Controlo de Acesso

Nesta secção apresentam-se alguns trabalhos desenvolvidos no âmbito da aplicação de controlo de acesso.

2.2.3.1ACADA

Em [Pereira, '12a] é proposta a ACADA (Access Control-driven Arquiteture with Dynamic Adaptation), uma arquitetura que permite incorporar objetos adaptáveis derivados de políticas de controlo de acesso. Neste paper são também referidos os problemas que ocorrem com a separação entre as políticas de controlo de acesso e os mecanismos que as aplicam:

- Uma utilização direta de API's (JDBC,ADO.net…) de um modo consistente com as políticas de controlo de acesso definidas, leva a que o utilizador tenha que ter um

domínio completo das políticas estabelecidas. Isto pode ser difícil à medida que a complexidade das políticas de controlo de acesso aumenta;

- Os programadores podem utilizar qualquer expressão sql nas API's, o que pode levar à violação das políticas de acesso estabelecidas na base de dados;
- A atualização das políticas de controlo de acesso leva a uma atualização dos mecanismos que as implementam. Não existindo maneira automática de traduzir as alterações nas API's correntes;
- Nalgumas partes do código é necessária a escrita de código *Hard-Coded* para gestão das restrições em tempo de execução. Não existindo maneira de atualizar este código de controlo de acesso escondido.

2.2.3.2 SELINKS

SELINKS [Corcoran, '09] é uma linguagem de programação baseada na construção de aplicações Web multi-camadas seguras. Esta fornece um modelo de programação uniforme, ao estilo de LINQ [Meijer, '06] e Ruby on Rails [Hansson, '04], com sintaxe de linguagem para acesso a objetos armazenados em servidores comuns ou em base de dados. Nesta linguagem, os programadores definem meta dados relevantes para a segurança (designados por rótulos) usando tipos algébricos e estruturados e definem também as funções de aplicação de políticas, que a aplicação chama explicitamente para mediar o acesso aos dados. Para garantir que a chamada às funções de aplicação de políticas não é deixada de fora ou não é feita incorretamente, o SELINKS implementa um sistema de tipos designado por FABLE. Este sistema assegura que os dados sensíveis nunca são acedidos diretamente sem consultar a função de aplicação de políticas apropriada. Estas funções correm num servidor remoto e verificam em tempo de execução as ações que um utilizador pode realizar, permitindo um controlo mais eficiente do que os SGBD-R são atualmente capazes de realizar. Na Figura 14 temos um exemplo de uma função que realiza pesquisa por texto em entradas na base de dados. Esta função recebe como parâmetros de entrada a credencial do utilizador *cred* e a frase de pesquisa *keyword.* A verificação da permissão do utilizador é realizada através da função de controlo de políticas *acess_str*. A função *acess_str* recebe como parâmetros: as credenciais do utilizador, o rótulo de segurança e o texto representando os dados protegidos. No caso de permissão aceite, é enviado o texto com os dados se não, nada é enviado.

```
1 fun getSearchResults(cred, keyword) server {
2   for (var row ← table_handle)
3   where (var txtOpt = access_str(cred, row.lab, row.text);
4             switch(txtOpt) {
5             case Just(data) → data  ~/.*{keyword}.*/
6             case Nothing → false
7             })
8    [row]
9 }
```

Figura 14 - Exemplo de função de aplicação de políticas retirada de [Corcoran, '09]

Este sistema permite um controlo mais eficiente do que os SGBD-R são atualmente capazes de realizar, mas os componentes no lado cliente não se adaptam automaticamente às alterações realizadas nas políticas no lado servidor, fazendo com que o utilizador tenha que dominar as políticas de controlo de acesso estabelecidas e os esquemas da base de dados. O que pode ser difícil com o aumento da complexidade das políticas.

2.2.3.3 JIF

Jif [Napes, '12] é uma linguagem de programação que estende o Java, para permitir o suporte para controlo de fluxo de informação e controlo de acesso. A aplicação das políticas é efetuada tanto em tempo de compilação como em tempo de execução. As políticas de controlo de acesso são aplicadas através da utilização de rótulos que expressam restrições em como a informação deve ser usada. Por exemplo considere a seguinte declaração da variável x:

```
int (Alice->Bob) x;
```

Neste caso, a política de segurança indica que x é controlada por Alice e que Alice permite que esta informação seja vista por Bob. Com anotações de rótulos como este, o compilador do JIF analisa o fluxo de informação entre programas para determinar se existe a aplicação tanto da confidencialidade como da integridade da informação. Embora o JIF aborde alguns aspetos importantes tais como, aplicação de políticas em tempo de compilação e de execução, uma das desvantagens da utilização do JIF reside no facto de os utilizadores apenas terem conhecimento de inconsistências após a execução do

compilador do JIF. Deste modo, as políticas de acesso não são conhecidas durante a fase de desenvolvimento, o que pode levar à criação de código que vá de encontro às permissões.

2.2.3.4 Aplicação de Controlo de Acesso Usando Vistas

Rizvi em [Rizvi, '04] apresenta um sistema de controlo de acesso que verifica se as consultas à base de dados contêm uma autorização apropriada. Esta abordagem requer que o administrador especifique vistas de segurança na base de dados para filtrar o conteúdo das tabelas. O sistema só executa a consulta na base de dados depois de verificar que esta pode ser executada contra uma vista filtrada da tabela. Este mecanismo de segurança é transparente, pois os utilizadores não autorizados não sabem que as suas consultas estão a ser testadas contra tabelas filtradas. Esta transparência nem sempre é desejada, pois quando existe um conjunto complexo de políticas em vigor, é importante explicar aos utilizadores porque é que as validações de autorização falharam. Isto permite que os utilizadores possam rever os seus pedidos em função das políticas estabelecidas, de modo a reescrever o pedido de acesso.

2.2.3.5 Modelo de Segurança Baseado na Adaptação Dinâmica

Em [Morin, '10] é proposta uma aproximação baseada em Modelos de segurança dinamicamente adaptáveis. O objetivo deste trabalho é a aplicação de mecanismos de controlo de acesso flexíveis que se adaptam dinamicamente às alterações realizadas nas políticas de controlo de acesso.

Foram descritos dois tipos de modelos independentes:

- O modelo arquitetural de negócios
- O modelo de segurança

A composição dinâmica do modelo de segurança a partir do modelo arquitetural permite que as políticas de segurança se adaptem de acordo com regras pré-definidas e também que lidem com as mudanças não planeadas. Isto é realizado da seguinte maneira:

O modelo de segurança possui um conjunto de regras de acesso e o contexto nas quais estas regras se aplicam. Em tempo de execução, um conjunto destas regras são selecionadas. Estas regras são depois compostas no modelo arquitetural de negócios, obtendo como resultado um modelo arquitetural que aplica políticas de segurança nas aplicações. Para processar as alterações às políticas de segurança, existe um sistema de monitorização. Aquando de uma alteração, o sistema de monitorização é despoletado e o modelo de segurança adaptado. A definição das políticas de segurança é realizada através do estabelecimento de ligações entre os componentes das diferentes camadas. Não foi

contemplada a forma como se aplicam as políticas de segurança a nível de software, um dos pontos essenciais da nossa arquitetura.

2.3 Abordagem Tecnológica

Neste capítulo são expostas as tecnologias usadas para o desenvolvimento da ADCA. As aplicações de base de dados desenvolvidas na arquitetura ADCA foram implementadas em Java [Oracle, '12f] e Sql Server 2008[Microsoft, '12m], tendo sido desenvolvida uma versão estática em C# [Microsoft, '12a]. Para facilitar o desenvolvimento das aplicações foram utilizadas as ferramentas de desenvolvimento NetBeans [Netbeans, '12] e Visual Studio [Microsoft, '12q]. O armazenamento das políticas de acesso é realizado com recurso à ferramenta Sql-Server [Microsoft, '12m] da Microsoft e a base de dados exemplo utilizada é a NorthWind [Microsoft, '12i]. São também apresentadas as principais características das transações, um elemento utilizado no desenvolvimento da ADCA.

2.3.1 Java

Java[Oracle, '12f] é uma linguagem de programação desenvolvida na Sun Microsystems (recentemente comprada pela Oracle). Esta é uma linguagem orientada a objetos com sintaxe muito parecida com as linguagens C e C++ mas oferecendo mais facilidades, tais como:

- Portabilidade: um programa java pode ser executado em qualquer sistema operativo, enquanto que em C e C++, é necessário a compilação diferente para cada SO;
- Criação de interfaces gráfica: em java o desenvolvimento de interfaces gráficas é mais facilitado que em C ou C++.

Na Figura 15 descreve-se a forma como é realizada a compilação e execução de programas Java. O código fonte dos programas é guardado em ficheiros de formato .java que depois são compilados em ficheiros .class.

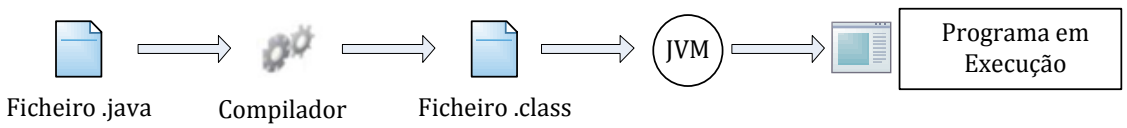


Figura 15 - Esquema de funcionamento da linguagem de programação JAVA

Estes ficheiros .class podem ser executados em muitos sistemas operativos, visto que a tecnologia utilizada para execução é uma máquina virtual, designada de JVM(Java Virtual Machine). Esta permite a execução em Microsoft Windows, Solaris OS, Linux e Mac OS, sendo assim uma linguagem que permite a execução num amplo conjunto de sistemas operativos. No contexto da linguagem de programação java destacamos a ferramenta de desenvolvimento NetBeans[Netbeans, '12] e as seguintes API's, entre outras:

- API JDBC: utilizada para interação com base de dados;
- API Reflection: permite a uma classe a sua própria inspeção em tempo de execução.

O NetBeans e estas API's são descritas com mais detalhe nos próximos parágrafos.

2.3.1.1NetBeans

Netbeans[Netbeans, '12] é um IDE Java desenvolvido pela empresa Sun Microsystems, que auxilia os programadores de *software* a escrever, compilar, realizar debug e instalação de aplicações. O seu foco é a linguagem de programação java mas suporta outras linguagens tais como: C, C++, Ruby, PHP, XML e HTMl. O Netbeans fornece também uma base sólida para desenvolvimento de projetos e módulos, pois possui um grande conjunto de bibliotecas e API's, deste modo facilitando muito o desenvolvimento de aplicações em Java.

2.3.1.2 JDBC

A JDBC (Java Database Connectivity) [Oracle, '12i] é uma API para a linguagem de programação java que permite o acesso e interação com um amplo conjunto de SGBDs. Esta fornece métodos para consulta e atualização de dados numa base de dados. Podemos criar expressões SQL e utilizá-las em código java, onde o resultado da sua execução é transformado em objetos fáceis de manipular.

A API JDBC permite:

- Estabelecer conexões com uma qualquer base de dados;
- Envio de instruções Sql para a base de dados;
- Obter e processar os resultados recebidos da base de dados em resposta das instruções enviadas.

Para utilização desta tecnologia é necessária a utilização de um driver JDBC, que é um adaptador no lado do cliente, que converte os pedidos de programas java para um protocolo que o SGBD consiga perceber. Na Figura 16 podemos encontrar uma arquitetura típica do JDBC, onde os comandos JDBC são transformados em operações no SGBD através da utilização de um driver compatível com o SGBD utilizado.

Os drivers JDBC são divididos em quatro categorias:

- JDBC-ODBC: É um driver de implementação de base de dados que implica o uso do driver ODBC [Microsoft, '12k] para ligação à base de dados. Este driver converte invocações JDBC em invocações a funções em ODBC;
- Drives com acesso para API nativa: Drivers que mapeiam as invocações JDBC para as invocações na API do cliente fazendo o carregamento do código binário em cada máquina cliente;
- Protocolos de rede: Drivers de implementação de base de dados que fazem uso de uma camada de *middleware* entre a execução do programa e a base de dados. Esta camada intermédia converte as chamadas JDBC direta ou indiretamente no protocolo específico usado pela base de dados;
- Protocolos nativos: Drivers de implementação de base de dados que convertem as invocações JDBC diretamente para o protocolo usado na base de dados. Enquanto neste caso a conversão é feita diretamente no cliente, no caso anterior, a conversão é feita na camada intermédia de *software*.

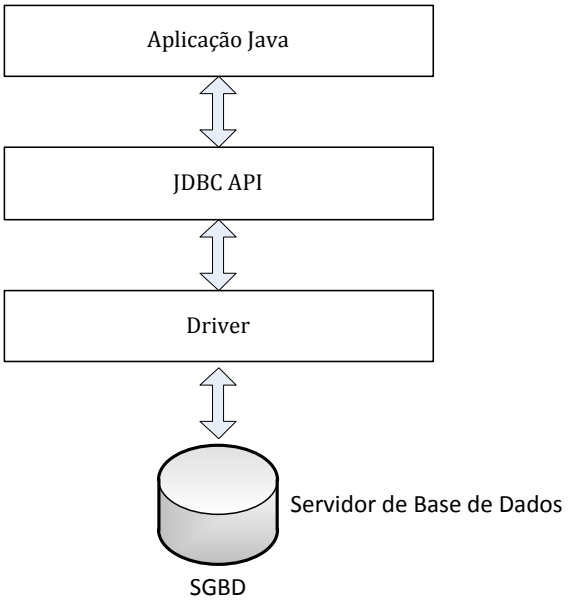


Figura 16 - Esquema de funcionamento da API JDBC

Após conseguir uma correta ligação à base de dados é necessário a utilização de objetos para comunicação com a base de dados (envio de expressões sql e receção de resultados), para tal fim, existem três tipos de objetos, os quais se encontram representados na Tabela 5.

Objeto	Descrição
Statement	Objeto usado para acesso geral à base de dados. Útil quando são usadas expressões sql estáticas em tempo de execução. A interface deste objeto não aceita parâmetros.
PreparedStatement	Objeto usado quando pretendemos utilizar uma expressão sql várias vezes, alterando apenas os parâmetros de entrada.
CallableStatement	Usado quando queremos aceder a stored procedures armazenadas na base de dados. A interface deste objeto aceita parâmetros de entrada em tempo de execução.

Tabela 5 - Objetos JDBC para interação com a base de dados

Ao executarmos objetos do tipo Statement e PreparedStatement com instruções sql do tipo Select é retornado um objeto com os dados resultantes. O objeto resultante desta execução tem o nome de *ResultSet* [Oracle, '12m]. A iteração pelos dados armazenados no *ResultSet* é realizada através dos métodos que se apresentam na Tabela 6.

Método	Descrição
absolute(n)	Move para a linha com o índice n.
afterLast()	Move o cursor para a próxima linha em relação à última linha do ResultSet.
first()	Move o cursor para a primeira linha do ResultSet.
last()	Move o cursor para a última linha do ResultSet.
next()	Move o cursor para a próxima linha do ResultSet.
previous()	Move o cursor para a linha anterior.
relative(n)	Move o cursor n linhas em relação à linha atual.
beforeFirst	Move o cursor para a linha anterior em relação à primeira linha do ResultSet.

Tabela 6 - Métodos de movimentação existentes em objetos ResultSet

Após estarmos na linha requerida podemos utilizar os métodos de leitura do JDBC para obter os valores guardados no ResultSet, nos quais indicamos o tipo de valor no nome do método e como parâmetro o nome da coluna. No caso da leitura de um inteiro que se encontra na coluna "Id", utilizamos o método getInt() e como parâmetro de entrada, a string "Id", identificando a coluna a ser lida. Para modificar os valores do ResultSet existem os seguintes métodos, descritos na Tabela 7.

Método	Descrição
Update___(n,X)	Atualiza a coluna n para o objeto de valor X.
UpdateRow()	Atualiza a base de dados subjacente com as alterações da linha corrente do ResultSet.
moveToInsertRow	Move o cursor para uma linha temporária, que permite a inserção de elementos.
insertRow	Insere o conteúdo da linha temporária no ResultSet e na base de dados.
DeleteRow	Apaga a linha da tabela que se encontra selecionada, do ResultSet e da base de dados subjacente.
moveToCurrentRow	Move o cursor para a linha anteriormente conhecida.

Tabela 7 - Métodos de atualização e inserção de campos de objetos ResultSet

Temos assim a descrição dos métodos necessários para manipulação de campos e objetos em base de dados utilizando a linguagem de programação java e a API JDBC.

2.3.1.3 Reflection

A Reflection API [Oracle, '12r] [Oracle, '12q]é geralmente usada por programas que requerem a habilidade de analisar e modificar o seu comportamento em tempo de execução. Reflection é uma técnica poderosa que permite a aplicações realizar operações, que de outra maneira seria impossível. Com a Reflection podemos [Green, '12]:

- Determinar a classe de um objeto;
- Obter informação sobre uma classe, como: campos, métodos, construtores e super classes;
- Encontrar que constantes e métodos pertencem a uma interface;
- Criar uma instância de uma classe, cujo nome era desconhecido, até à execução da aplicação;

- Obter e modificar o valor de um campo de um objeto, mesmo que o nome deste campo seja desconhecido pela aplicação até a sua execução;
- Invocar um método de um objeto desconhecido até execução da aplicação.

Esta técnica não deve ser usada discriminadamente pois apresenta um conjunto de desvantagens [Oracle, '12q]:

- *Overhead* de *Performance*: Esta técnica envolve tipos que são resolvidos dinamicamente e por consequente, algumas operações de otimização por parte da *Java Virtual Machine* não poderão ser realizadas, devendo então ser evitado o seu uso em secções do código que são chamadas frequentemente;
- Restrições de Segurança: A Reflection requer permissões em tempo de execução que poderão não estar presentes se estiver a ser executada ao abrigo de um gestor de segurança. Esta é uma importante consideração para código que necessita de ser executado num contexto restrito de segurança, como por exemplo um Applet;
- Exposição de entidades privadas: A Reflection permite a execução de operações que seriam ilegais em código não refletivo, como aceder a campos privados. Esta possibilidade pode resultar em código disfuncional, podendo mesmo destruir a portabilidade deste.

2.3.2 C#

C# [Microsoft, '12a] é uma linguagem de programação desenvolvida no ano 2000 para construção de uma vasta gama de aplicações que correm na plataforma .NET [Microsoft, '12h]. Pertence à família das linguagens derivadas do c, como c++ e java, mas ao contrário do c++, esta é uma linguagem inteiramente orientada a objetos. Sendo esta mais recente que java, apresenta diversas melhorias, tornando esta uma linguagem, mais robusta e fácil de utilizar para desenvolvimento de aplicações. Algumas das vantagens de C# em relação a Java são[Microsoft, '12b]:

- Integração profunda no sistema operativo Windows, facilitando por exemplo a interação com ferramentas como o Sql Server;
- Utilização da instrução foreach, que permite uma iteração facilitada em arrays de elementos, garantido que todos os elementos são analisados;
- C# incorpora arrays multidimensionais ao contrário de java.

No contexto da linguagem de programação C# utilizamos a ferramenta de desenvolvimento VisualStudio e a API ADO.Net que é utilizada para interação com base de

dados. Esta API, tal como a ferramenta de desenvolvimento VisualStudio, são descritas com mais detalhe nos próximos parágrafos.

2.3.2.1 VisualStudio

Visual Studio.NET[Microsoft, '12q] é um IDE da Microsoft que oferece um conjunto poderoso de ferramentas para desenvolvimento e debug de aplicações. Permite-nos ainda trabalhar dados com o SQLServer de uma maneira simples, sem necessidade de abrir qualquer interface de base de dados, contemplando assim os nossos requisitos.

2.3.2.2 ADO.Net

O ADO.NET [Microsoft, '12l] consiste num conjunto de classes definidas pela *framework* .NET que podem ser utilizadas para acesso a base de dados remotas. O ADO.NET oferece tecnologia para conectar a base de dados, executar comandos e obter resultados. Estes resultados são processados diretamente e colocados em objetos *DataSet* [Microsoft, '12d]. Este objeto possui um conjunto de tabelas e as relações entre elas. Na Figura 17 podemos ver a principal arquitetura do ADO.net e os principais componentes que a compõem. Nesta arquitetura existem dois componentes principais:

- O fornecedor de dados da plataforma .NET (*DataProviders*): Constituído por objetos de manipulação e leitura de dados:
    - *Connection*: fornece conectividade a qualquer SGBD;
    - *Command*: permite a execução de expressões sql; execução de Stored-Procedures e envio ou receção de informação;
    - *DataReader*: fornece uma stream de dados de alta-performance com origem na base de dados;
    - *DataAdapter*: fornece uma ponte entre o objeto *DataSet* e a base de dados. O *DataAdapter* utiliza objetos do tipo *Command* para executar comandos na base de dados, tanto para preencher o *DataSet* com dados como para reconciliar as alterações realizadas no *DataSet* de volta à base de dados.
- O *DataSet* contém uma coleção de um ou mais objetos *DataTable* preenchidos com os resultados da execução de uma instrução Select.

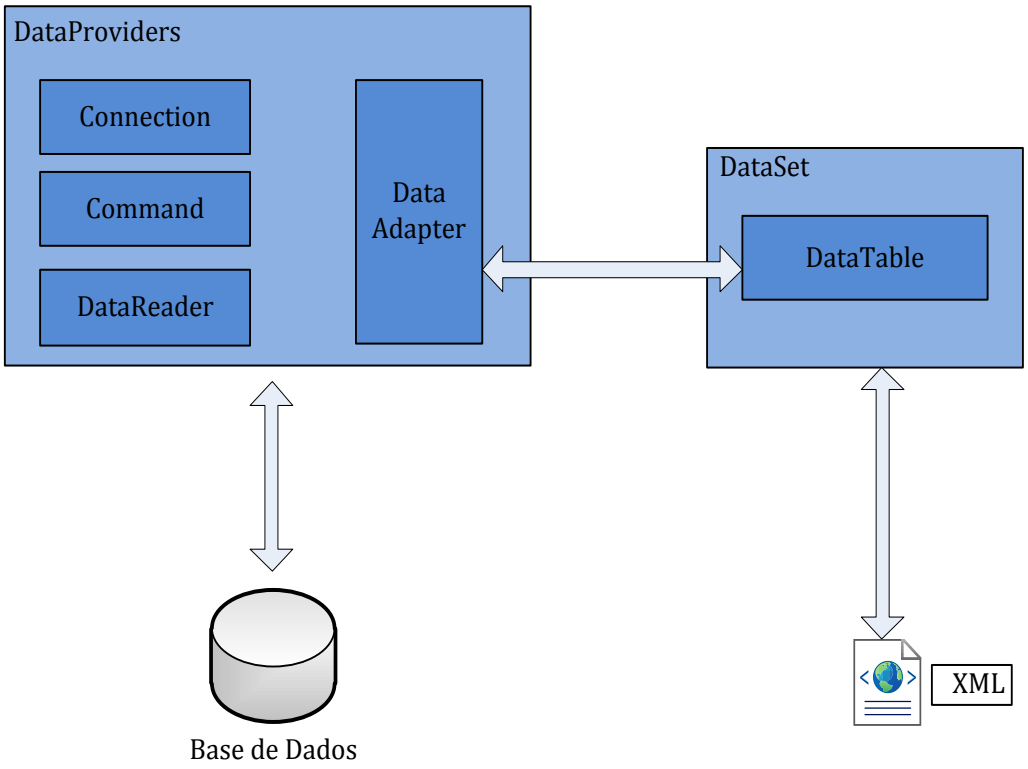


Figura 17 - Principais componentes do Ado.Net

No exemplo da Figura 18 evidencia-se a criação de um comando sql do tipo Select utilizando o ADO.net. Considere a existência de uma variável do tipo String com o nome "crud", onde está armazenada uma expressão sql do tipo Select. Neste exemplo, primeiro é executado o método *SelectCommand* do objeto *DataAdapter* (variável *da*), com a respetiva instrução sql (objeto String designado por *crud)*. Em seguida, preenche-se o *DataSet* (variável *ds*) com os elementos resultantes do comando sql. E por fim a primeira tabela obtida é passada para um objeto *DataTable* (variável *dt*) que pode ser manipulada como uma tabela normal em C#.

```
da.SelectCommand = new SqlCommand(crud, conn,ss.getTransaction());
cursor = 1;
ds.Clear();
da.Fill(ds);
dt = ds.Tables[0];
nRows = dt.Rows.Count;
```

Figura 18 - Exemplo de utilização de Ado.Net

2.3.3 Transações

Uma transação [Java2s, '12] [Microsoft, '12o] é uma sequência de operações realizadas como uma unidade singular de trabalho. que deve possui quatro propriedades:

- Atómica: todas as modificações são realizadas ou nenhuma é realizada;
- Consistente: quando concluída, uma transação deve deixar todos os dados num estado consistente;
- Isolada: modificações realizadas por transações concorrentes devem deixar todos os dados num estado consistente;
- Durável: depois de completa, os seus efeitos ficam permanentemente no sistema, mesmo depois de uma falha no sistema.

Para controlo das transações, existem quatro níveis de isolamento, os quais se descrevem na Tabela 8 [Oracle, '12o].

Tipo de Isolamento	Descrição
ReadUncommited	Este é o menor nível de isolamento. Permite a uma transação aceder a alterações que ainda não foram confirmadas por outras transações. Deste modo as transações não são isoladas umas das outras.
ReadCommited	Neste caso, é criado um lock para as operações write, até ao fim da transação, ou seja a transação só pode ler os dados de tabelas que já foram comited (enviados). Se uma transação quiser atualizar uma tabela, tem que obter um lock, e outras transações terão que esperar pela libertação do lock para realizar as suas alterações.
RepeatableRead	Neste modo o lock é aplicado a todos os dados usados pela transação, tanto a leitura como a escrita.
Serializable	Os locks são colocados em tabelas, fazendo-se lock de tabelas completas. Este é o nível de maior isolamento.

Tabela 8 - Descrição dos níveis de isolamento em Transações

Em bases de dados onde coexistem várias transações concorrentes é originado um conjunto de ocorrências que permitem a leitura de dados inconsistentes. Essas ocorrências são as seguintes:

- *Phantom Read*: Ocorre quando dentro de uma mesma transação, a expressão sql é executada duas vezes e o segundo resultado inclui linhas que não eram visíveis na primeira. Como se pode ver pelo exemplo da Figura 19, com o nível de isolamento *ReadCommited*, a transação 1 executa a mesma expressão duas vezes. Mas na segunda execução, uma linha adicional que não existia na primeira execução aparece, visto que uma segunda transação executa uma inserção entre as duas leituras;

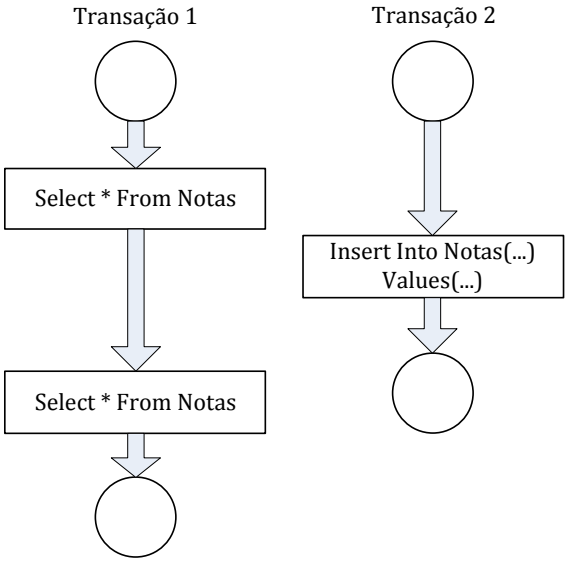


Figura 19 - Exemplo de Phantom Read

- *Dirty Read*: Ocorre quando uma transação está a ler dados e outra transação está concorrentemente a atualizá-los. Se o tipo de isolamento for *ReadUncommited*, a primeira transação lê os dados que ainda não foram *commited* (enviados). Pelo exemplo da Figura 20, podemos verificar que quando a primeira transação altera o atributo nota para 10 e como o isolamento é do tipo *ReadUncommited*, a transação 2 faz leitura dos dados *uncommited*, neste caso nota 10, tendo de seguida a transação 1 ter sido rejeitada, o que faz com que haja uma leitura de um valor incorreto;
- *Non-Repeatable Read*: Ocorre quando numa mesma transação, a mesma expressão possui resultados diferentes. Isto ocorre quando uma segunda transação atualiza os dados usados por outra transação. Como podemos ver pela Figura 21, com o isolamento ReadCommited, a transação 1 executa duas leituras, mas a segunda resulta em resultados diferentes, visto que a transação 2 atualiza os valores da tabela entre as duas leituras da primeira transação.

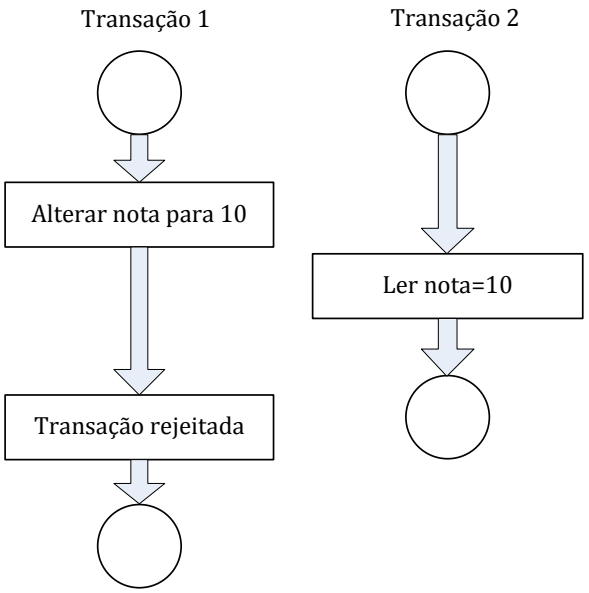


Figura 20 - Exemplo de dirty-read

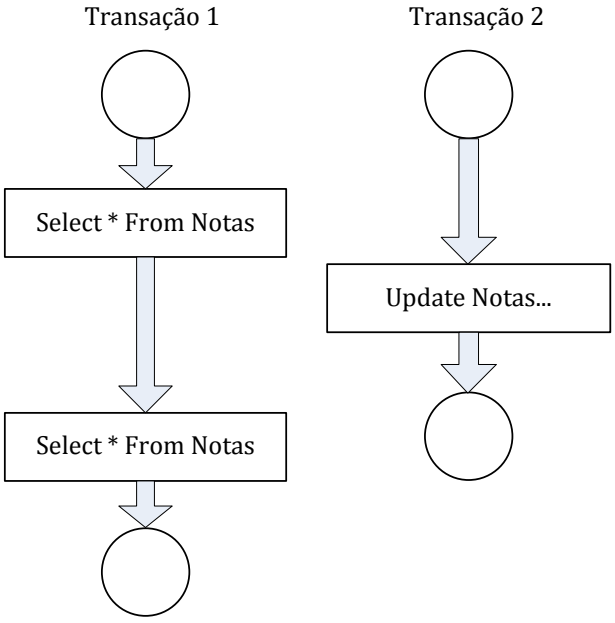


Figura 21 - Exemplo de Non-Repeatable Read

Na Tabela 9 descrevem-se para que níveis de isolamento ocorrem estes problemas.

Nível Isolamento	Dirty-Read	Non-Repeatable Read	Phantom-Read
ReadUncommited	Sim	Sim	Sim
ReadCommit	Não	Sim	Sim
RepeatableRead	Não	Não	Sim
Serializable	Não	Não	Não

Tabela 9 - Tabela com a ocorrência de incorreções nos vários níveis de isolamento

3 Arquitetura Dinâmica de Controlo de Acesso

Neste capítulo apresentamos a ADCA e fazemos a descrição dos vários componentes que a compõem.

3.1 Arquitetura Geral

A ADCA é formada por um conjunto de componentes criados com o propósito de facilitar a interação com base de dados (como as oferecidas por exemplo por API's, como JDBC [Oracle, '12i], ADO.Net [Microsoft, '12l], JPA [Oracle, '12j],LINQ [Microsoft, '12g] e Hibernate [JBOSS, '12]) oferecendo ao mesmo tempo controlo de acesso a informação sensível. Este controlo de acesso é realizado através de mecanismos de controlo de acesso incorporados na camada de negócio do cliente, sendo que estes devem estar de acordo com as políticas de controlo de acesso definidas num servidor de políticas.
O modelo arquitetural da aplicação ADCA está dividido em duas partes:

- Lado Cliente: local onde o acesso dos utilizadores à informação é controlada por mecanismos de controlo de acesso;
- Lado Servidor: local onde são armazenadas as políticas de controlo de acesso que regulam os mecanismos de controlo acesso evidenciados no ponto anterior.

Ao contrário do modelo PDP-PEP, onde por cada tentativa de acesso de um comando no lado PEP é enviado um pedido ao PDP para verificação da permissão de execução, na ADCA, a decisão sobre o acesso é tomada ao nível do lado cliente. Neste existem mecanismos de controlo de acesso, construídos com base nas políticas de controlo de acesso estabelecidas num servidor de base de dados, designado por servidor de políticas. Deste modo, quando existem alterações nas políticas armazenadas neste servidor, os mecanismos de controlo de acesso adaptam-se dinamicamente, permitindo que estejam sempre de acordo com as políticas estabelecidas.

O lado cliente disponibiliza um conjunto de serviços regulados por controlo de acesso que permitem uma fácil interação com base de dados. O lado Cliente foi desenvolvido usando Java. Para demonstrar que os métodos disponibilizados pelos serviços de negócio podem ser implementados usando várias tecnologias, foi desenvolvida uma versão estática em C# do lado do Cliente.

Por outro lado, o lado Servidor contém um conjunto de políticas de acesso armazenadas num Servidor Sql Server, que descriminam quais os serviços que podem ser utilizados pelos utilizadores em tempo de execução. A adaptação dinâmica às alterações

efetuadas nas políticas de acesso é possível devido à implementação de um monitor que monitoriza as políticas. Quando ocorre uma alteração é despoletado um *trigger* que permite que o monitor de políticas comunique as alterações efetuadas. Permitindo assim uma correta adaptação da lógica de negócio de modo a refletir as políticas de acesso estabelecidas. A Figura 22 apresenta a arquitetura geral desenvolvida para a resolução do problema.

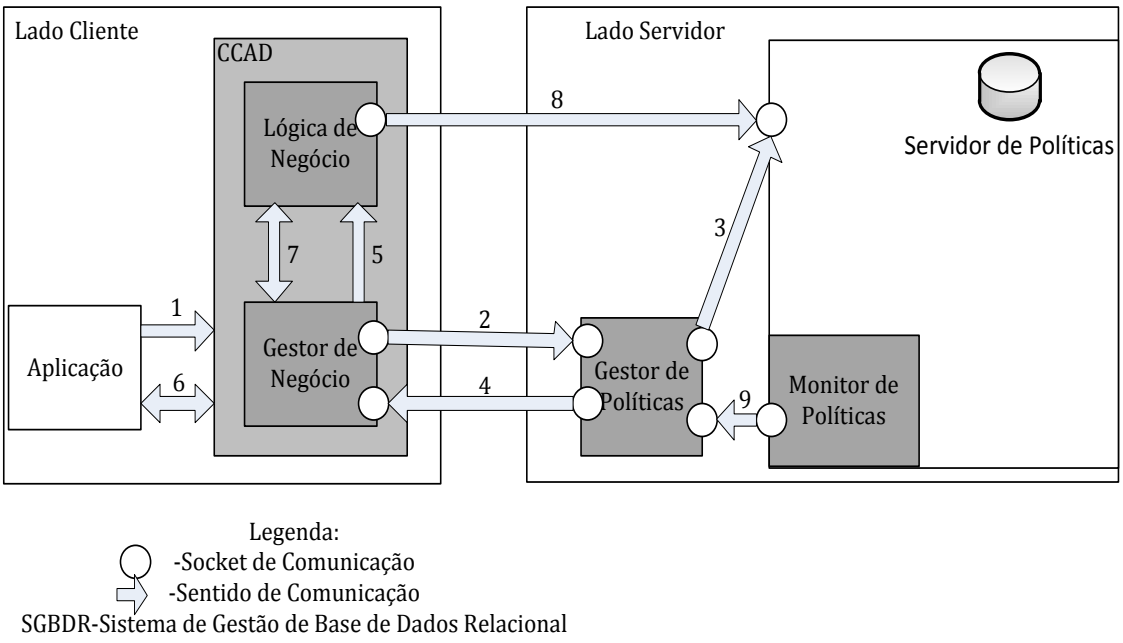


Figura 22 - Arquitetura Geral Desenvolvida

No lado do cliente existe o CCAD (Componente de Controlo de Acesso Dinâmico), este componente é constituído por dois sub-componentes principais:

- Lógica de Negócio: neste componente são armazenados um conjunto de esquemas de negócio e a sua respetiva implementação. Os esquemas de negócio referem-se a um conjunto de interfaces que disponibilizam os métodos que permitem ao utilizador interagir com a base de dados relacional;
- Gestor de Negócio: este componente permite gerir a lógica de negócio de acordo com as políticas definidas e possibilitar a instanciação das implementações dos esquemas de negócio aos quais se dá o nome de serviços de negócio. A identificação do utilizador e a gestão de sessões são também geridos pelo gestor de negócio.

O CCAD adapta-se às alterações efetuadas nas políticas em tempo de execução, permitindo assim um sistema robusto e seguro.

No lado do Servidor existe uma base de dados implementada em Sql Server, onde as políticas de controlo de acesso estão armazenadas. Visto que o lado cliente é independente do controlo de acesso utilizado no armazenamento das políticas, poderíamos utilizar um qualquer controlo de acesso. A nossa escolha recaiu sobre o controlo de acesso baseado em papéis (RBAC). Esta escolha deveu-se essencialmente à facilidade concedida por este modelo na gestão de controlo de acesso, facilitando a atribuição de permissões por parte dos administradores do sistema. O modelo RBAC selecionado para construção do servidor de políticas foi o modelo RBAC1 (descrito em 2.1.3) que permite a existência de, sessões, autorizações, utilizadores e um conjunto de papéis numa estrutura do tipo hierárquica. O modelo desenvolvido é detalhado no ponto 3.3.3.2.2. Este modelo permite associar a papéis, esquemas de negócio e as respetivas expressões CRUD (expressões Sql do tipo Create Read Update Delete, baseadas em [Pereira, '10; Pereira, '11b]).

Toda a informação de e para o CCDA passa pelo gestor de políticas, sendo assim, o intermediário entre o CCAD e o servidor de políticas. Nos serviços de negócio instanciados pelo gestor de negócios são utilizadas expressões Sql do tipo CRUD.

O modo de operação deste sistema é o seguinte:

- A camada de aplicação cria instâncias do CCAD (Figura 22-1) onde se atribui a autenticação do utilizador e da aplicação, tanto para identificação na base de dados relacional como para identificação com o servidor de políticas;
- O CCAD estabelece a conexão com o gestor de políticas (Figura 22-2) e tenta fazer o login do utilizador no sistema;
- O gestor de políticas liga-se ao servidor (Figura 22-3) e autentica-se a si próprio se ainda não estiver autenticado, autenticando também o CCAD;
- O gestor de políticas "recolhe" as políticas (Figura 22-3) do servidor de políticas de acordo com o utilizador e a aplicação;
- O gestor de políticas liga-se ao gestor de negócios e envia as políticas de controlo de acesso (Figura 22-4);
- O CCAD cria automaticamente a lógica de negócio;
- A camada de aplicação faz então pedidos ao CCAD (Figura 22-6) para gerir a execução de expressões CRUD;
- O gestor de negócios contacta a lógica de negócios (Figura 22-7) para execução dos pedidos da camada de aplicação. Em seguida, a lógica de negócio contacta o

servidor relacional (Figura 22-8) e executa a instrução CRUD, sendo os resultados da sua execução retornados à aplicação;

- Sempre que existam alterações nas políticas armazenadas no servidor de políticas, o monitor de políticas envia a informação sobre as alterações para o gestor de políticas (Figura 22-9). Este comunica-as ao gestor de negócios (Figura 22-4) que se adapta dinamicamente de modo a estar consistente com as políticas de acesso estabelecidas.

Em[Pereira, '12a] apresentam-se os problemas relativos à utilização corrente de API's (JDBC e ADO.net) com o estabelecimento de políticas de controlo de acesso. Nestas API's a aplicação de políticas de acesso impõe uma separação nítida entre as políticas de controlo de acesso e os mecanismos responsáveis pela sua aplicação. Isto leva a ocorrência de várias desvantagens, tais como:

- Uma utilização direta destas API's consistentemente com as políticas de acesso definidas levaria a que o utilizador tivesse que ter um domínio completo das políticas estabelecidas. Isto poderia ser difícil à medida que a complexidade das políticas de controlo de acesso aumentava. Este problema é resolvido na ADCA com a utilização da lógica de negócio que se adapta dinamicamente em tempo de execução às políticas estabelecidas e apresenta apenas os serviços permitidos;
- Utilização de expressões CRUD que podem ir de encontro às políticas de acesso definidas. Para resolução deste caso foram definidas no servidor de políticas as expressões CRUD a que o utilizador tem acesso consoante os esquemas de negócio, levando a um maior controlo destas.

3.2 Comunicação entre os Componentes da ADCA

Neste capítulo apresentam-se as estruturas das mensagens que são trocadas entre os componentes que compõem a nossa Arquitetura.

3.2.1 Comunicação Gestor de Negócios – Gestor de Políticas

Nos próximos parágrafos, são descritas as soluções encontradas para a troca de informações entre gestor de negócio e o gestor de políticas, permitindo uma correta construção e adaptação da lógica de negócio.

3.2.1.1 Login no Gestor de Políticas

A primeira ação de um utilizador antes de qualquer pedido ao servidor, é a realização do login no sistema, de modo a este ser identificado no sistema e receber as permissões a que tem direito. Com a instanciação do gestor de negócios, este autentica-se no sistema. A autenticação é realizada através da troca de duas mensagens. A primeira mensagem sinaliza o gestor de políticas que o cliente pretende efetuar o login. Esta mensagem tem a estrutura representada na Figura 23.

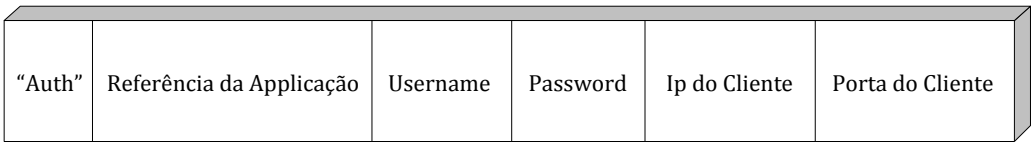


Figura 23 - Estrutura da mensagem de *login*

Esta estrutura possui os seguintes campos:

- O primeiro campo corresponde à string "AUTH" e identifica a mensagem como sendo uma mensagem de autenticação;
- A referência da aplicação permite ao gestor de políticas identificar qual a aplicação que o utilizador pretende utilizar e verificar se possui permissão para a utilizar;
- Os campos Username e Password permitem a identificação do utilizador no sistema, e são usados para verificar a sua correta identidade;
- Os campos Ip e Porta do cliente servem para permitir localizar na rede o gestor de negócios correspondente ao utilizador. Esta informação é guardada no servidor de políticas permitindo a comunicação das alterações realizadas às políticas de controlo de acesso estabelecidas.

Depois de verificada a correta identidade do utilizador e do seu acesso à aplicação, é necessário criar uma sessão para este novo cliente. Para tal existe uma tabela sessão, cuja criação é referida no ponto 3.3.3.2.2 e na qual se cria uma nova entrada com o ip e porta do cliente respetivo. Com a identidade verificada e a correspondente sessão criada, o gestor de políticas sinaliza o gestor de negócios através de uma mensagem. Esta corresponde a uma de duas Strings:

- "OK", Identidade verificada;

- "NOK", dados de autenticação errados, ou acesso negado à aplicação.

No caso de receção da uma mensagem a negar o acesso, a comunicação é terminada entre o gestor de negócios e o gestor de políticas.

3.2.1.2 Comunicação das Informações sobre Políticas

No caso de receção de mensagem afirmativa por parte do gestor de negócios, aquando do *login*, é agora necessário obter as informações sobre quais os esquemas de negócio a que o utilizador tem acesso para a correta construção da lógica de negócio. Para tal, o gestor de negócios envia a mensagem "getBus" a requerer o envio de informação sobre os esquemas de negócio a que tem direito. O Gestor de Políticas adquire a informação sobre os esquemas de negócio através do servidor de políticas seguindo os passos especificados mais à frente no ponto 3.3.3.3.1 e transmite-a seguindo a troca de mensagens representada na Figura 24.

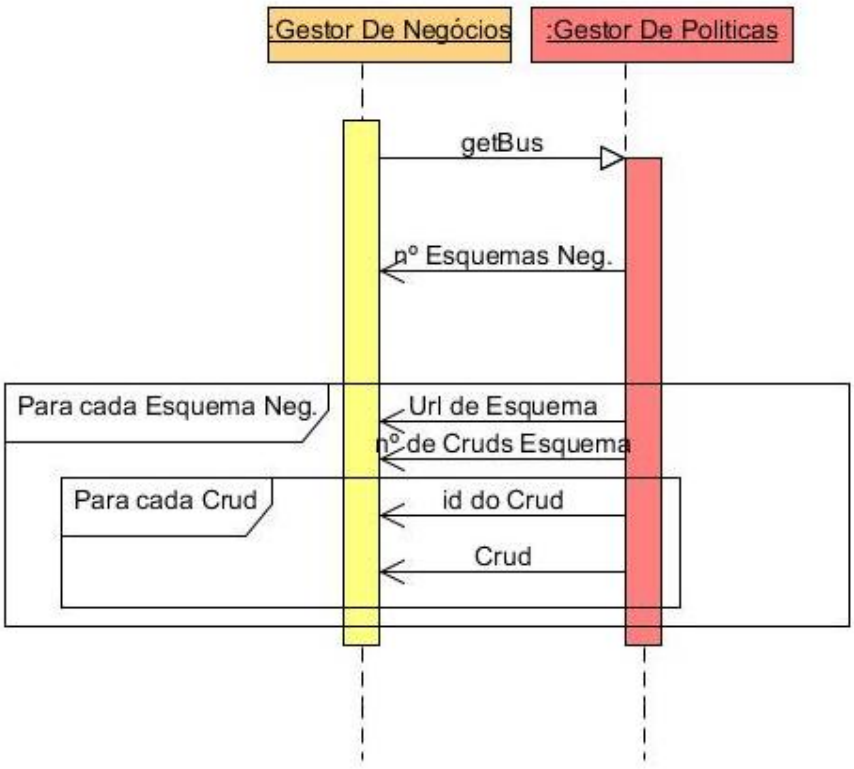


Figura 24 - Conjunto de mensagens trocadas, para obtenção de esquemas de negócio e CRUDS com acesso

Este processo decorre de acordo com os seguintes passos:

- O gestor de negócios, envia a mensagem "getBus" para informar o gestor de políticas que pretende obter informação sobre quais os esquemas de negócio e expressões CRUD a que tem acesso;
- O gestor de políticas adquire todas as informações sobre as permissões do cliente e envia o número de esquemas de negócio, ao qual tem direito;
- Por cada esquema de negócio é enviado por parte do gestor de políticas, o Url, para identificação da interface principal referente ao esquema de negócio e o número de expressões CRUD correspondentes a este;
- Por cada expressão CRUD correspondente, é enviado o *id* identificando a expressão CRUD e a respetiva String.

Com este conjunto de passos, são enviadas as informações sobre as permissões a que o utilizador do ADCA tem direito. O gestor de negócios pode então criar a respetiva lógica de negócio.

3.2.1.3 Comunicação das Alterações Efetuadas nas Políticas

A comunicação entre o monitor de políticas e o gestor de políticas é despoletada por *triggers* existentes na base de dados, aquando da alteração de políticas no sistema, este monitor é especificado em mais detalhe no ponto 3.3.5. Quando ocorrem alterações nas políticas do sistema, o monitor de políticas envia para o gestor de política a informação sobre as alterações que ocorreram. Depois de sinalizado pelo monitor de políticas, o gestor de negócios obtém as informações necessárias aos gestores de negócios afetados pela alteração, através de algoritmos especificados mais à frente nos pontos 3.3.3.4.1 e 3.3.3.4.2. Esta informação é enviada para o gestor de negócios que possui uma thread à escuta de alterações realizadas nas políticas. A mensagem enviada do gestor de políticas ao gestor de negócios tem o seguinte formato, apresentado na Figura 25.

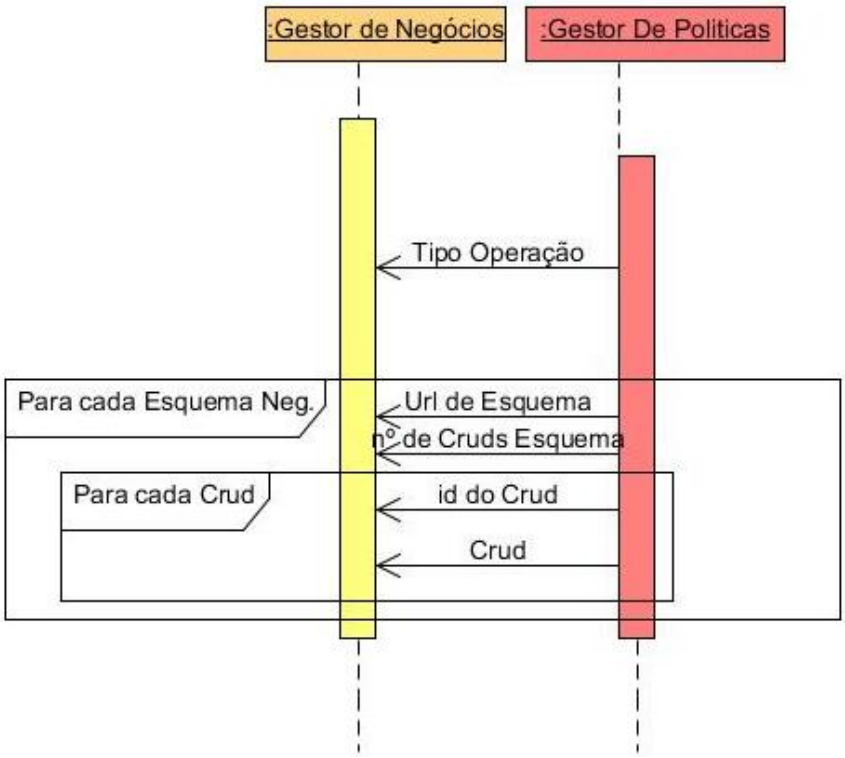


Figura 25 - Conjunto de mensagens trocadas, para informação sobre alterações efetuadas às políticas de controlo de acesso

Este esquema tem como base o esquema referido no ponto 3.2.1.2. Apenas com a alteração que o gestor de negócios não envia nenhuma mensagem, este recebe a mensagem referida como Tipo de Operação, evidenciado na Figura 25, que contém uma *String* a designar o tipo de operação.

Esta *String* pode conter dois valores:

- "Add": para identificar operação de adição de esquemas de negócio;
- "Remove": para identificar operação de remoção de esquemas de negócio.

Desta maneira sinaliza-se o gestor de negócio das alterações efetuadas, sendo depois a lógica de negócio adaptada dinamicamente às respetivas alterações.

3.2.2 Comunicação Monitor de Políticas – Gestor de Políticas

Quando são alteradas as políticas de acesso, é enviada a respetiva informação para o gestor de políticas, através do monitor de políticas. Esta informação tem que identificar que tipo de alteração ocorreu (Remoção ou Inserção de novas políticas), o tipo de

permissão (Autorização ou Delegação) e ainda informação sobre o utilizador e papéis associados. A mensagem enviada para o gestor de políticas contém os seguintes campos:

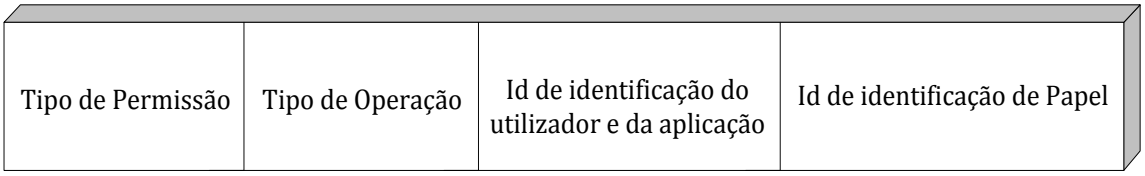


Figura 26 - Conjunto de campos referentes às mensagens enviadas do monitor de políticas ao gestor de políticas

O primeiro campo identifica o tipo de permissão atribuída ao papel (autorização ou delegação); No segundo campo existe o tipo de operação (se operação de adicionar, ou de remoção); Os restantes campos servem para identificar o utilizador, a aplicação e também o papel associado com esta operação.

Chegamos assim, neste capítulo a um conjunto de estruturas de mensagens que permitem a comunicação entre os vários componentes.

3.3 Desenvolvimento da ADCA

Neste capítulo apresenta-se os vários componentes que compõem a arquitetura apresentada no ponto 3.1 e as estratégias utilizadas para o seu desenvolvimento.

3.3.1 Lógica de Negócio

A lógica de negócio é constituída por um conjunto de esquemas de negócio. Estes são constituídos por interfaces que disponibilizam os métodos necessários para o utilizador interagir com a base de dados relacional. A gestão da lógica de negócio é realizada pelo gestor de negócios, que permite que a lógica de negócio se adapte dinamicamente em tempo de execução, consoante as políticas de acesso que se encontrem definidas no nosso servidor de políticas

Esquemas de Negócio

A maioria das API's existentes para envio de instruções sql para a base de dados (JDBC, ADO.Net, JPA) fornecem comandos em que a sintaxe não é validada e não têm em conta as políticas de controlo de acesso e sua constante alteração [Pereira, '12a]. O nosso

objetivo com a construção dos esquemas de negócio é desenvolver um conjunto de interfaces que depois, com a sua implementação (Serviços de Negócio), possamos obter um objeto que forneça apenas os métodos que estão de acordo com as políticas. Para tal, foram construídos um conjunto de interfaces baseadas nos papers [Pereira, '11a; Pereira, '12a; Pereira, '12b], que permitem a execução de expressões CRUD (Create, Read, Update, Delete) em base de dados relacionais. A este conjunto de interfaces dá-se o nome de esquemas de negócio. Podemos então afirmar que os esquemas de negócio em conjunto com as expressões CRUD são os elementos chaves para utilização de API's (JDBC e Ado.net) para interação com base de dados relacionais. Na Figura 27 encontramos um caso de alteração de um campo de uma tabela utilizando ADO.net e JDBC, que são as API's utilizadas neste trabalho para interação com a base de dados. Verifica-se que a forma como se desenvolve é relativamente diferente, o que levaria um programador de *software* a ter que aprender diferentes API's, logo implicando uma curva substancial de aprendizagem. Com vista à redução da curva de aprendizagem do programador de *software*, os esquemas de negócio contêm os nomes dos métodos iguais quer para a sua implementação em estática C# quer para a sua implementação dinâmica em Java. No caso da utilização dos esquemas de negócio desenvolvidos neste trabalho, este conjunto de instruções expresso na Figura 27 seria executado por apenas dois comandos principais: um para execução da instrução CRUD e outro para alteração do campo correspondente.

```
//ADO.net
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = new SqlCommand(crud, conn);
SqlCommandBuilder cb = new SqlCommandBuilder(da);
DataSet ds = new DataSet();
ds.Clear();
da.Fill(ds);
dt = ds.Tables[0];
dr["Tnumero"] = value;
cb.GetUpdateCommand();
da.Update(ds);

//JDBC
Statement st = conn.createStatement();
ResultSet rs=st.executeQuery(sql);
rs.next();
rs.updateInt("Tnumero", value);
rs.updateRow();
```

Figura 27 - Conjunto de instruções necessárias para a alteração de um campo de uma tabela em ADO.net e JDBC

Com base nestes aspetos, foi desenvolvido o seguinte esquema principal representativo dos esquemas de negócios presente na Figura 28.

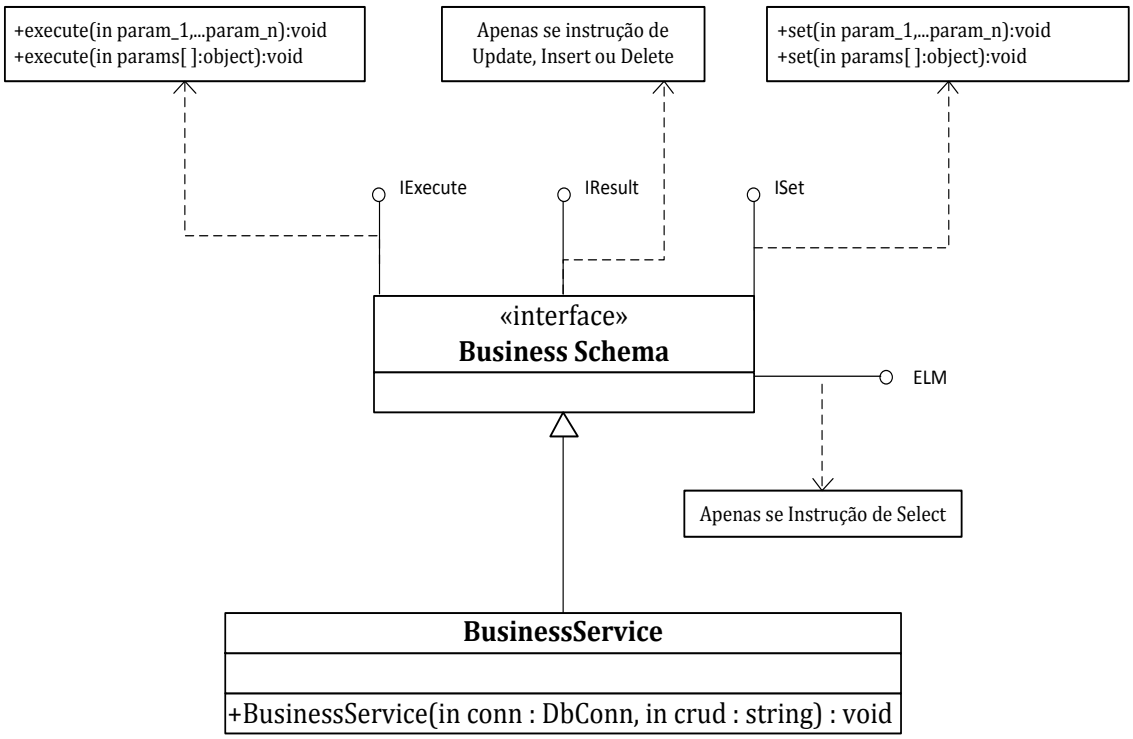


Figura 28 - Modelo principal dos esquemas de negócio

Podemos verificar que todos os esquemas de negócio são constituídos por um método construtor, designado por *BusinessService*. Este método permite a instanciação de um serviço de negócio, que é um objeto que fornece métodos para o utilizador interagir com a base de dados relacional. Este método recebe uma conexão para a base de dados (SGBD-R, onde os métodos são executados) designada por conn, do tipo *Connection* e também por uma String que representa a expressão CRUD que vai ser executada. Este método é transparente ao utilizador e apenas é chamado pelo gestor de negócios para a instanciação dos serviços de negócio, sendo a expressão CRUD utilizada, uma das expressões a que o utilizador tem acesso. As expressões CRUD são definidas no servidor de políticas pelo administrador, sendo atribuídas aos respetivos esquemas de negócio. Aquando da instanciação de um serviço de negócio o utilizador seleciona qual a expressão CRUD a utilizar com base no *id* respetivo desta expressão que se encontra armazenada no servidor de políticas. No modelo exposto, a interface IResult já se encontra criada por defeito, sendo adicionada ao conjunto de esquemas de negócio dependendo do tipo da

instrução CRUD. As outras interfaces são criadas manualmente pelo Administrador ou pelo programador de *software* para possibilitar uma correta utilização dos esquemas.

Interface IExecute

Uma interface obrigatória em todos os esquemas de negócio é a interface IExecute. Esta interface é constituída por métodos *execute* que permitem a execução da expressão CRUD na base de dados relacional (SGBD-R) aquando da sua chamada. Muitas vezes é conveniente a utilização de parâmetros nas expressões Sql. A vantagem de utilização de expressões Sql que recebem parâmetros é que esta pode ser fornecida com valores diferentes de cada vez que queiramos executá-la. Para exemplo, considere o caso em que queremos obter os elementos da tabela Pessoa que tenham as idades 10 e 30.

Neste caso podemos executar duas expressões sql:

"Select * From Pessoa Where Idade=10"
"Select * From Pessoa Where Idade=30"

Este tipo de utilização pode ser simplificado com a utilização de parâmetros em expressões Sql. Por exemplo, as expressões sql passadas a um objeto no JDBC podem ter a seguinte estrutura:

"Select * From Pessoa Where Age=?"

Onde o sinal de pontuação ? indica a posição do parâmetro ao qual vai ser atribuído um valor. Deste modo evita-se a criação de expressões Sql repetidas, onde apenas alguns valores são alterados. Para tal definimos métodos execute que possuem parâmetros de entrada, os quais correspondem a parâmetros estabelecidos nas instruções CRUD. Para esta situação foram estabelecidas duas aproximações:

- Aproximação fechada: temos que explicitar os vários parâmetros que são atribuídos ao CRUD na chamada ao execute:

```
void execute(int id) throws SQLException;
```

Neste método de exemplo tínhamos que explicitar o valor do *id*, que depois seria atribuído ao parâmetro definido na instrução CRUD.

- Aproximação aberta: definimos um objeto que contém os valores dos vários parâmetros:

```
void execute(Object[] obj) throws SQLException;
```

Neste caso, criámos um array de objetos antes da execução com os vários elementos instanciados.

Apresentamos agora um caso de exemplo de utilização de métodos do tipo execute. Considere a seguinte expressão CRUD que seleciona as entradas da tabela Pessoa com base na cidade e idade:

"select * from Pessoa Where Cidade=? and Idade=?"

Neste caso o administrador ou o programador de *software* podem criar o seguinte método:

```
void execute(String cidade,int idade);
```

Este método permite a atribuição de valor ao campo cidade e ao campo referente à idade.

Interface ISet

A interface ISet foi desenvolvida para estabelecer os parâmetros de instruções CRUD antes da sua execução na base de dados relacional. Esta interface tal como a interface IExecute possui duas aproximações:

- Aproximação fechada:

```
void set(int id);
```

Neste caso definíamos o valor do id, antes da execução.

- Aproximação aberta:

```
void set(Object[] obj);
```

Neste caso podemos criar um array de objetos antes da execução.

Como caso de exemplo de utilização, considere a seguinte expressão CRUD para inserção de uma nova entrada na tabela Pessoa:

"insert into Pessoa(Nome,Cidade,Idade) VALUES(?,?,?)"

Para a resolução deste caso podemos definir na interface ISet o seguinte método:

```
void set(String nome,String cidade,int idade);
```

Este método permite a atribuição de valores aos parâmetros especificados na expressão Crud antes da sua execução.

Interface IResult

A interface IResult é obrigatória nas instruções CRUD do tipo Insert, Update e Delete. Esta possui um único método designado por *getNAffectedRows* que retorna o número de linhas que foram alteradas pela execução deste tipo de instruções.

No caso de expressões CRUD do tipo Select, foi criada uma estrutura designada por ELM (Estrutura Lógica de Memória) que contém um conjunto de interfaces que permitem manipular facilmente os resultados da execução da expressão na base de dados relacional. A descrição das interfaces que compõem esta estrutura é realizada nos próximos parágrafos.

3.3.1.1 Definição da Estrutura Lógica de Memória (ELM)

No caso de execução de uma expressão CRUD do tipo Select é retornada uma tabela de dados com as linhas e colunas resultantes da execução dessa instrução na base de dados relacional. Esta tabela é transformada em objetos, que podem ser manipulados na linguagem de programação utilizada. No caso do java a transformação ocorre para um objeto do tipo *ResultSet* (JDBC) e em ADO.net para um objeto do tipo *DataSet*. Os tipos presentes nas tabelas resultantes da execução das instruções CRUD são automaticamente transformados pela camada de *software* intermédia utilizada (JDBC e ADO.Net) em tipos existentes na linguagem de programação (Por exemplo: o tipo nvarchar em SQL Server é convertido no tipo String em java e c#, mapeamento pode ser analisado em [Oracle, '12k]). Para possibilitar a interação do utilizador com esta tabela foram criados métodos por

defeito para movimentação na tabela, leitura, inserção, atualização de elementos e também para possibilitar a remoção de linhas. Estes métodos compõem a estrutura designada por estrutura lógica de memória (ELM), que foi criada com o objetivo de ser robusta e fácil de utilizar, sendo o nome dos próprios métodos intuitiva o suficiente, para dar ao utilizador a entender a sua funcionalidade. As políticas de controlo de acesso do tipo obrigatório estão presentes nesta estrutura lógica, visto que na construção das interfaces que a compõe, podemos definir para cada campo a permissão de leitura, atualização, inserção e a permissão de remoção de linha. Inicialmente utilizamos uma interface designada por IWrite que nos permitia definir para que objetos da base de dados tínhamos permissão de escrita. Estas permissões eram usadas tanto para operações de update como para operações de insert, mas esta aproximação era demasiado extensa e possibilitava um menor controlo sobre a autorização dos campos. Para obter um maior controlo sobre as estas operações, alterámos o esquema de modo a que estes campos fossem declarados nas interfaces IUpdate se tivesse permissão de atualização de campos ou na interface IInsert, se tivesse permissão de inserção do elemento. Chegámos assim ao esquema representado pela Figura 29 que fornece métodos úteis para manipulações de instruções Select e se aplica ao mesmo tempo políticas de controlo de acesso úteis para proteger dados confidenciais.

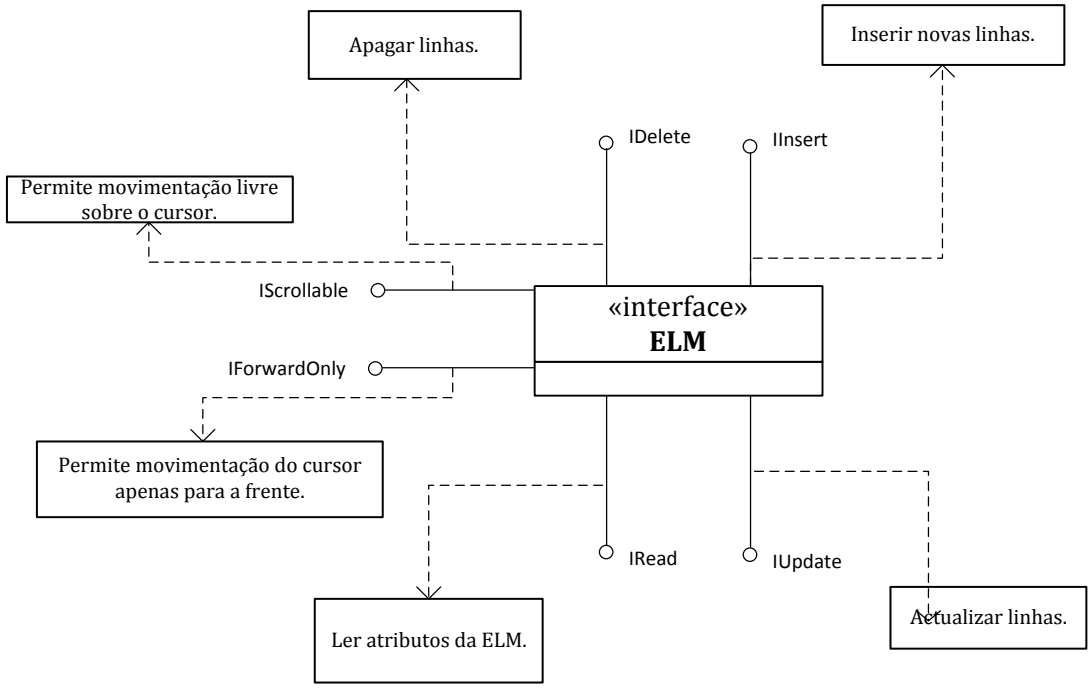


Figura 29 - Esquema representativo da Estrutura Lógica de Memória (ELM)

Esta estrutura é constituída pelas seguintes interfaces:

- IRead: define os atributos que podem ser lidos;
- IUpdate: define os atributos que podem atualizados;
- IInsert: define os atributos que podem ser inseridos;

  Foram ainda definidas as interfaces para movimentação da tabela;
- IScrollable : define se a movimentação pelos dados é realizada livremente;
- IForwardOnly: define se a movimentação pelos dados é realizada apenas para a próxima linha da tabela.

Nos próximos parágrafos apresentam-se os métodos que compõem estas interfaces e exemplos de utilização das mesmas.

3.3.1.1.1 Movimentação dentro da ELM

Para movimentação dentro da tabela resultante da execução da expressão Crud do tipo Select, existem duas aproximações:

- Livre: o cursor pode ser movido para qualquer posição dentro da tabela;
- Restrita: o cursor pode apenas ser movido para a próxima linha da tabela.

Deste modo, na definição do esquema de negócio podíamos estender uma interface com uma destas aproximações para movimentação do cursor, oferecendo mais controlo das operações. Para tal, foram criadas duas interfaces:

- Interface IScrollable: segue a aproximação livre e fornece métodos (Descritos na Tabela 10) para um controlo livre do cursor;
- Interface IForwardOnly: segue a aproximação restrita e fornece apenas um método, para movimentação do cursor para a linha seguinte, designado de `boolean moveNext()`.

Com a utilização dos métodos fornecidos por estas interfaces, podemos colocar o cursor de seleção na posição requerida de modo a podermos executar de seguida métodos que manipulam os campos da linha selecionada.

Método	Descrição
`boolean moveNext()`	Permite o movimento do cursor, para a próxima linha da tabela, retorna verdadeiro, se foi possível mover, ou falso se não.
`boolean moveAbsolute(int pos)`	Permite o movimento do cursor para uma posição absoluta, estabelecida, pelo utilizador, retorna verdadeiro, se foi possível mover ou falso, se não.
`boolean moveRelative(int offset)`	Permite o movimento do cursor para uma posição relativa em relação à posição atual, estabelecida, por um *offset* definido, pelo utilizador, retorna verdadeiro, se foi possível mover ou falso, se não.
`void moveBeforeFirst()`	Move o cursor para a posição anterior à primeira linha da tabela.
`boolean moveFirst()`	Move o cursor para a primeira linha da coluna, retorna verdadeiro, se foi possível mover ou falso, se não.
`void moveAfterLst()`	Move o cursor para a linha a seguir à última linha da tabela.
`boolean moveLast()`	Move o cursor para a última linha da coluna, retorna verdadeiro, se foi possível mover ou falso, se não.

Tabela 10 - Descrição dos métodos da interface IScrollable

3.3.1.1.2 Leitura de Campos da ELM

A leitura dos valores dos campos que compõem a ELM é realizada pela interface IRead, na qual são definidos o nome dos elementos e o respetivo tipo. Como exemplo prático, considere a existência de uma tabela Aluno com o esquema da Figura 30. Considere também a seguinte expressão CRUD :

"Select * FROM Aluno"

Esta expressão permite obter todos os campos da tabela Aluno. A execução desta expressão CRUD permite obter uma tabela com o valor dos campos *Id*, *Nome, Idade e Morada*, sendo que os tipos são automaticamente convertidos para a linguagem de

programação utilizada pelas API's de interação com a base de dados (JDBC e ADO.net). Podemos verificar que os campos *Id e idade* são do tipo int em java e c# e os campos *nome* e morada têm o tipo String (Para conversão de tipos ver [Oracle, '12k]).

Figura 30 - Tabela exemplo Aluno

A permissão para leitura dos campos da ELM é realizada a partir da Interface IRead. Neste caso, o administrador ou o programador de *software* procediam à criação dos seguintes campos na interface IRead para permitir a sua leitura:

```
int rId();
String rNome();
int rIdade();
String rMorada();
```

Estes métodos possuem como retorno o tipo de dados do campo, e como designação a junção da vogal r (para distinguir operações, representa métodos que permitem a leitura de campos) com o nome do campo a ser lido. Quando os esquemas de negócio fossem adicionados à lógica de negócio pelo gestor de negócio, seria automaticamente criada a implementação destes métodos, que permitem a leitura do valor do campo da linha da tabela onde o cursor se encontra. Assim, com a interface IRead podemos discriminar quais os campos que podem ser lidos. No caso anterior teríamos permissão para leitura de todos os campos da tabela Aluno.

3.3.1.1.3 Inserção de Campos na ELM

A interface IInsert é constituída por métodos de apoio à inserção de elementos na ELM e também pelos campos que podem ser inseridos. Os métodos de gestão do protocolo de inserção de campos (Ver Tabela 11) são sempre obrigatórios nesta interface. No início da inserção é chamado o método *beginInsert* que move o cursor para uma entrada onde existe uma linha temporária com os campos correspondentes da tabela. Estes campos podem ser alterados com a execução dos métodos de inserção de elementos.

Os métodos utilizados para inserção de elementos recebem como parâmetro um objeto com um tipo correspondente ao tipo do campo corretamente convertido para a linguagem de programação utilizada, enquanto a designação do método começa pela vogal i (designando a inserção de elemento) em junção com o nome do campo.

Método	Descrição
`void beginInsert()`	Permite o início da inserção, move o cursor para a linha temporária de inserção.
`void endInsert(boolean moveToPreviousRow)`	Permite a finalização da inserção dos elementos e as alterações são repercutidas na base de dados, o utilizador pode ainda definir se o cursor volta à posição antes da inserção ou se mantém a posição atual.
`void cancelInsert()`	Permite cancelar a inserção de elementos, não havendo alterações à tabela.

Tabela 11 - Descrição dos métodos da interface IInsert

Tomando como exemplo a tabela da Figura 30 e a seguinte expressão CRUD:

Select * FROM Aluno,

Para inserção de uma nova linha com os campos *Id* e *Nome*, criávamos os seguintes métodos na interface IInsert:

```
int iId();
String iNome();
int iIdade();
String iMorada();
```

Estes métodos permitem inserir o valor dos campos na nova linha temporária, sendo as alterações (criação de uma nova linha) repercutidas para a base de dados, com a execução do método endInsert. Um exemplo de utilização dos métodos de inserção encontra-se mais à frente no ponto 3.3.1.3.

3.3.1.1.4 Atualização de Campos da ELM

A interface IUpdate é constituída por métodos (também obrigatórios) de apoio a atualização de elementos da tabela (ver Tabela 12). Neste caso, a definição dos métodos de atualização é semelhante à da interface IInsert. A única mudança é a alteração da vogal inicial i para u, designando uma operação de update:

```
int uId();
String uNome();
int uIdade();
String uMorada();
```

O início da atualização dos campos é realizado com a execução do método *beginUpdate*. Os métodos criados pelo utilizador (*uId* e *uNome*), permitem a alteração do valor do campo da linha selecionada. As alterações são apenas realizadas com a execução do método *updateRow*, podendo estas ser canceladas com execução do método *cancelUpdate*. Um exemplo de utilização dos métodos de atualização da ELM encontra-se mais à frente no ponto 3.3.1.3.

Método	Descrição
`void beginUpdate()`	Inicializa o início da atualização da linha selecionada da tabela.
`void updateRow()`	Permite a atualização da tabela e a repercussão das alterações para a base de dados.
`void cancelUpdate()`	Permite cancelar a atualização de elementos da linha selecionada.

Tabela 12 - Descrição dos métodos da interface IUpdate

3.3.1.1.5 Remoção de Linhas da ELM

A remoção de linhas é permitida, se for realizada a extensão à interface IDelete (Interface já criada por defeito). Esta é constituída pelo método *deleteRow()* que permite apagar uma linha da tabela. Quando executada, a linha da tabela selecionada pelo cursor é eliminada e as alterações repercutidas na base de dados relacional.

3.3.1.2 Desenvolvimento dos Métodos Referentes aos Esquemas de Negócio

Os métodos desenvolvidos para os esquemas de negócio utilizam *software* intermédio para interação com a base de dados, nomeadamente, JDBC e ADO.net. Enquanto no Java a implementação destes é realizada de uma forma dinâmica em tempo de execução, no C# foi criada uma implementação estática para demonstrar que estes métodos podem ser implementados utilizando várias tecnologias. Nos próximos parágrafos descrevem-se as estratégias utilizadas para a implementação destes métodos, utilizando as API's JDBC e ADO.net.

3.3.1.2.1 Implementação dos Métodos Construtores

Em java, no qual se utiliza a API JDBC, o método construtor permite a criação de um objeto *PreparedStatement* através da execução do método *prepareStatement* do objeto *Connection*. A utilização do objeto *PreparedStatement* deve-se a duas vantagens em relação a objetos *Statement* [Oracle, '12l]:

- Na maioria dos casos a expressão CRUD é imediatamente enviada para o SGBD, onde é compilada. Isto permite que com a sua chamada a instrução seja apenas executada e não compilada novamente, poupando tempo de execução;
- Este objeto permite a execução de expressões CRUD com ou sem parâmetros, abrangendo assim todas as expressões CRUD que possam ser utilizadas.

O construtor em JDBC é criado de forma diferente para expressões do tipo Select ou para expressões IUD (Insert, Update e Delete). Na Figura 31 temos a implementação do método de construção de um serviço de negócio em JDBC para expressões do tipo Select.

```
public Cat_s(Connection conn,String crud) throws SQLException {
    this.conn=conn;
    this.crud=crud;
    ps=conn.prepareStatement(crud,ResultSet.TYPE_SCROLL_SENSITIVE,
                             ResultSet.CONCUR_UPDATABLE);

}
```

Figura 31 - Implementação do método construtor em JDBC

Nesta figura podemos verificar que guardamos tanto a conexão, como a expressão CRUD na nossa ELM. Em seguida criámos um objeto do tipo *PreparedStatement* que

permite a execução de instruções CRUD na base de dados. O método utilizado para a construção deste objeto designa-se de *prepareStatement* e tem como parâmetros de entrada:

- A variável *crud* contendo a expressão CRUD;
- O tipo de movimentação possível dentro do *ResultSet*: neste caso especificamos para todas as expressões *Select* o tipo *"TYPE_SCROLL_SENSITIVE"*, que permite uma movimentação livre dentro do *ResultSet*. Este tipo de movimentação é restringida se estendermos a interface *IForwardOnly*, pois apenas permite o movimento para a próxima linha;
- O modo do *ResultSet*: permite indicar se apenas se pode efetuar a leitura de campos ou se estes podem ser atualizados. Neste caso utilizamos o modo *"CONCUR_UPDATABLE"*, permitindo tanto a leitura como a atualização.

Na Figura 32 encontramos a implementação do construtor para expressões IUD. Neste caso, a única diferença encontra-se na forma como o objeto *PreparedStatement* é criado. Em expressões IUD não é necessário um *ResultSet*, pois não existe retorno de dados do SGBD. Como tal, omitimos os parâmetros para definir o modo de funcionamento do *ResultSet* e passamos apenas como parâmetro a variável crud referente à expressão CRUD a ser executada na base de dados.

```
public Cat_i(Connection conn,String crud) throws SQLException {
    this.conn=conn;
    this.crud=crud;
    ps=conn.prepareStatement(crud);
}
```

Figura 32 - Implementação do método construtor em JDBC para expressões IUD

Na API ADO.net o método construtor é criado através da utilização de um objeto *SqlDataAdapter*, que permite a execução de comandos e de um objeto *DataSet* para armazenamento dos resultados obtidos da base de dados. Na Figura 33 encontramos a implementação do método construtor para ADO.net. Tal como no caso do JDBC, guardamos a conexão para a base de dados (variável *cnctn*) e a expressão Crud (variável *crud*). Em seguida criámos dois objetos, o objeto *ds* do tipo DataSet e o objeto *da* do tipo SqlDataAdapter. Estes objetos permitem a interação com a base de dados, sendo a implementação do construtor neste caso, igual quer para expressões CRUD do tipo Select quer para expressões CRUD do tipo IUD (Insert, Update, Delete).

```
public Cat_s(SqlConnection cnctn, String crud)
{
    this.conn = cnctn;
    this.crud = crud;
    ds = new DataSet();
    da = new SqlDataAdapter();
}
```

Figura 33 - Implementação do Construtor em ADO.net

3.3.1.2.2 Implementação dos Métodos de Execução de Expressões CRUD

A execução das expressões CRUD na base de dados é realizada pela chamada ao método execute presente na interface IExecute. A implementação deste método é relativamente simples em JDBC. Para a execução de instruções CRUD do tipo IUD (Insert, Update, Delete), utiliza-se o método executeUpdate, o qual permite a execução da instrução CRUD na base de dados relacional e a obtenção do número de linhas alteradas (Figura 34).

```
public void execute() throws SQLException {
    nrows=ps.executeUpdate();
}
```

Figura 34 - Implementação do método execute para instruções IUD em JDBC

No caso de instruções do tipo Select é realizada a execução do método executeQuery. Este método permite a execução da expressão Select na base de dados e a obtenção dos resultados através de um objeto ResultSet.

```
public void execute() throws SQLException {
    rs=ps.executeQuery();
}
```

Figura 35 - Implementação do método execute para instruções Select em JDBC

Como podemos verificar pela Figura 35, o objeto *ResultSet* obtido através da execução do método executeQuery é guardado na ELM através da variável *rs*. No caso de um método execute com parâmetros de entrada e para estabelecer valores na expressão CRUD, utilizamos o método set do objeto PreparedStatement (Figura 36).

```
public void execute(int id) throws SQLException {
    ps.setInt(1,id);
    rs=ps.executeQuery();
}
```

Figura 36 - Implementação do método execute para expressões com parâmetros dinâmicos em JDBC

Como podemos verificar pela Figura 36, colocamos o valor dos parâmetros na expressão CRUD através da utilização dos métodos *set* do JDBC, antes da execução da expressão na base de dados. No caso do ADO.net, utiliza-se a Classe *SqlDataAdapter*(*da*) para execução da expressão CRUD. Para instruções CRUD do tipo IUD a utilização do objeto *SqlDataAdapter* é diferente:

- Para inserção utiliza-se a propriedade *InsertCommand*.
- Para atualização utiliza-se a propriedade *UpdateCommand*.
- Para apagar linhas utiliza-se a propriedade *DeleteCommand*.

A utilização destas propriedades em conjunção com o método *ExecuteNonQuery,* permite a execução da expressão IUD na base de dados e a obtenção do número de entradas afetadas. Na Figura 37 encontramos um exemplo de implementação para uma expressão do tipo *Insert*. Neste caso utiliza-se a propriedade *InsertCommand*, à qual atribuímos o nosso comando que contém a expressão CRUD (da.InsertCommand=Command). Em seguida executamos a expressão na base de dados com o método *ExecuteNonQuery*.

```
public void execute(String categoryName, String Description, Byte[] Image)
{
    SqlCommand command = new SqlCommand(crud, conn);
    da.InsertCommand = command;
    da.InsertCommand.ExecuteNonQuery();
}
```

Figura 37 - Implementação do método execute para expressões Insert em ADO.net

No caso de expressões *Select* utilizamos a propriedade *SelectCommand* do objeto *SqlDataAdapter* e preenchemos o *DataSet* com os dados obtidos, sendo depois utilizado um objeto *DataTable* para permitir uma melhor manipulação dos dados (Figura 38).

```
public void execute()
{
    da.SelectCommand = new SqlCommand(crud, conn);
    cb = new SqlCommandBuilder(da);
    cursor = 1;
    ds.Clear();
    da.Fill(ds);
    dt = ds.Tables[0];
    nRows = dt.Rows.Count;
}
```

Figura 38 - Implementação do método execute para expressões Select em ADO.net

Em ADO.net, a definição de expressões CRUD com parâmetros é diferente. Neste caso substitui-se o ? por @nome, sendo nome o nome do atributo.

Exemplo:

JDBC: Select * from Products where ProductId=?

ADO.net: Select * from Products where ProductId=@id

No caso de um método execute com parâmetros de entrada e para estabelecer valores na expressão CRUD, utilizamos uma expressão Regex [Zytrax, '12] para obter o nome dos atributos, de modo a podermos efetuar a atribuição de valores.

```
public void execute(int id)
{
    SqlCommand command = new SqlCommand(crud, conn);

    string[] words = Regex.Split(this.crud, @"(@\S\w*)");
    foreach (String s in words)
    {
        if (s.StartsWith("@") == true)
            command.Parameters.AddWithValue(s, id);

    }

    da.SelectCommand = command;
    cb = new SqlCommandBuilder(da);
    cursor = 1;
    ds.Clear();
    da.Fill(ds);
    dt = ds.Tables[0];
    nRows = dt.Rows.Count;
}
```

Figura 39 - Implementação do método execute com parâmetros em ADO.net

Na Figura 39 encontramos uma implementação exemplo de um método execute com parâmetros em ADO.net. Inicialmente criamos o objeto *SqlCommand* com a expressão CRUD, seguindo-se da atribuição do valor aos vários parâmetros através do método *AddWithValue* da propriedade *Parameters*. O resto da implementação é igual à implementação dos métodos execute sem parâmetros.

3.3.1.2.3 Implementação dos Métodos para Movimentação dentro da ELM

A implementação dos métodos para movimentação em ambas as API's é direta. Em JDBC utilizamos diretamente os métodos existentes nesta API para movimentação no ResultSet (Next,Last,etc…). No caso do ADO.net utilizamos uma variável inteira, designada de cursor, que contém o índice da linha selecionada. Este índice é alterado consoante o método chamado.

3.3.1.2.4 Implementação dos Métodos para Leitura de Campos da ELM

Os métodos para leitura de valores do campo são facilmente implementados, tanto para ResultSet(JDBC), como no DataSet(ADO.net). No caso do JDBC utilizamos um dos métodos get do ResultSet para obtenção do valor (Figura 40), enquanto no ADO.net obtemos o valor de uma linha relativa (dr) da DataTable (onde os dados estão armazenados), especificando o nome da coluna que contém o valor a que queremos aceder (Figura 41).

```
public int rvalor() throws SQLException{
   return rs.getInt("valor");}
```

Figura 40 - Implementação do método de leitura do valor de um campo em JDBC

```
public int rvalor()
{
    return (int)dr["valor"];

}
```

Figura 41 - Implementação do método de leitura do valor de um campo em ADO.net

3.3.1.2.5 Implementação dos Métodos de Atualização de Campos da ELM

Os métodos para escrita de valores em campos tanto para operações de atualização como para operações de inserção são implementados através da utilização do comando Update em JDBC (Figura 42) e da simples atribuição de valores a uma linha (DataRow) da DataTable em ADO.Net (Figura 43).

```
public void uvalor(int valor) throws SQLException{
    rs.updateInt("valor",valor);
}

public void ivalor(int valor) throws SQLException{
    rs.updateInt("valor",valor);
}
```

Figura 42 - Implementação dos métodos de escrita em JDBC

```
public void ivalor(int valor)
{
    dr["valor"] = valor;
}

public void uvalor(int valor)
{
    dr["valor"] = valor;
}
```

Figura 43 - Implementação dos métodos de escrita em ADO.net

3.3.1.2.6 Implementação dos Métodos Obrigatórios da Interface IUpdate

A atualização de elementos é realizada através da utilização da interface *IUpdate* da ELM, que contém métodos especializados para este tipo de operações. O método *beginUpdate* permite sinalizar o início de uma operação de atualização e apenas contém instruções para sinalização de estado.

O método *updateRow* realiza a atualização dos dados alterados e a repercussão dos mesmos para a base de dados relacional. No JDBC utiliza-se o método *updateRow* do objeto *ResultSet* (Ver Figura 44). No ado.net é utilizado o método *Update* do objeto *DataAdapter* (Ver Figura 45).

```
public void updateRow() throws SQLException {
    rs.updateRow();}
}
```

Figura 44 - Implementação do método updateRow em JDBC

```
public void updateRow()
{
    da.Update(dt);
    updaterow = false;
}
```

Figura 45 - Implementação do método updateRow em ADO.net

O método cancelUpdate permite cancelar as alterações entretanto realizadas. No JDBC é utilizado o método *cancelRowUpdates* e no ADO.net é utilizado o método *RejectChanges*. Estes métodos permitem rejeitar as alterações realizadas ao *ResultSet* e *DataSet* respetivamente.

3.3.1.2.7 Implementação dos Métodos Obrigatórios da Interface IInsert

A inserção de elementos é realizada através da utilização da interface IInsert da ELM, que contém métodos especializados para este tipo de operações. O método *beginInsert* permite a movimentação do cursor para uma linha temporária da tabela. No caso do JDBC, utilizamos o método *moveToInsertRow* do objeto *ResultSet*, que trata da criação e movimentação do cursor para essa linha No caso do ADO.net, temos que adicionar uma nova linha à *DataTable*. Estes dois casos encontram-se expostos na Figura 46 e Figura 47 respetivamente.

```
public void beginInsert() throws SQLException {
    rs.moveToInsertRow();
}
```

Figura 46 - Implementação do método beginInsert em JDBC

```
public void beginInsert()
{
    insertrow = true;
    dt.Rows.Add(new DataRow[1]);
    dr = dt.Rows[nRows];
    arDr[0] = dr;
}
```

Figura 47 - Implementação do método beginInsert em ADO.net

O método *endInsert* permite a inserção de elementos no *ResultSet* e *DataSet* e a repercussão das alterações para a base de dados. Este método tem ainda um parâmetro de entrada (um booleano) que permite indicar se é necessário voltar à posição original do ELM, antes do início da inserção.

```
public void endInsert(boolean moveToPreviousRow) throws SQLException {
    rs.insertRow();
    if(moveToPreviousRow)
        rs.moveToCurrentRow();
}
```

Figura 48 - Implementação do método endInsert em JDBC

```
public void endInsert(bool moveToPreviousRow)
{
        da.Update(arDr);
        nRows++;
        insertrow = false;
        if (moveToPreviousRow == true)
        {
            if ((cursor != -1) & (cursor < nRows))
                dr = dt.Rows[cursor];
            else dr = null;
        }
        else dr = dt.Rows[nRows - 1];
    }
```

Figura 49 - Implementação do método endInsert em ADO.net

Na Figura 48 temos a implementação do método *endInsert*, no caso de utilização da API JDBC. O método *insertRow* é utilizado para inserção de nova linha no *ResultSet* e na base de dados, enquanto o método *moveToCurrentRow* é utilizado para voltar à última posição conhecida antes da movimentação para a linha temporária. No caso do ADO.net (Ver Figura 49), utilizamos o comando *update* do Objeto *DataAdapter*, que recebe como

parâmetro a nova linha para inserção. No caso de ser necessária movimentação para a última linha conhecida, utilizamos a variável cursor que guarda o último índice de linha conhecido.

O cancelamento das alterações realizadas pela inserção é realizado de maneira diferente para JDBC e ADO.net. No caso do JDBC, basta a movimentação para a última linha conhecida (utilizando o método *moveToCurrentRow*), visto que a linha de inserção é uma linha temporária que não afeta o *ResultSet*.

No caso do ADO.net, temos que eliminar a última linha adicionada e voltar à última linha selecionada que se encontra na variável cusor (Figura 50).

```
public void cancelInsert()
{
    if (insertrow)
    {
        dt.Rows.RemoveAt(nRows);
        insertrow = false;
        dr = dt.Rows[cursor];
    }
}
```

Figura 50 - Implementação do método cancelInsert em ADO.net

3.3.1.3 Exemplos de Utilização dos Esquemas de Negócio

Neste capítulo apresentamos exemplos práticos para melhor demonstrar os modelos desenvolvidos nos pontos anteriores.

Considere a existência de uma tabela Notas conforme o esquema da Figura 51, que contém o nome e a nota de um aluno, e considere também a existência de três entidades: administrador, professor e aluno.

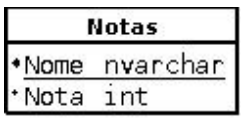


Figura 51 – Tabela de exemplo de caso prático Notas

As permissões para o Administrador, Professor e Aluno estão representadas na Tabela 13, Tabela 14 e Tabela 15 respetivamente.

Campo	Inserção	Atualização	Leitura	Apagar
Nome	Sim	Sim	Sim	Sim
Nota	Sim	Sim	Sim	

Tabela 13 - Tabela de permissões para Administrador

Campo	Inserção	Atualização	Leitura	Apagar
Nome	Não	Não	Sim	Não
Nota	Não	Sim	Sim	

Tabela 14 - Tabela de permissões para Professor

Campo	Inserção	Atualização	Leitura	Apagar
Nome	Não	Não	Sim	Não
Nota	Não	Não	Sim	

Tabela 15 - Tabela de permissões para Aluno

Neste caso é necessária a construção de três esquemas de negócio:

- Nota_admin para o caso do administrador;
- Nota_professor para o caso do professor;
- Nota_aluno para o caso do aluno;

Construção dos esquemas de negócio

Vamos apresentar nos próximos parágrafos os esquemas da ELM utilizados para resolução do problema proposto. O esquema da ELM para a entidade Nota_admin está representado na Figura 52. Como podemos verificar esta é a única entidade que possui autorização para apagar uma linha da tabela. Como tal é estendida a interface IDelete, permitindo ao administrador ter acesso ao método deleteRow. As interfaces IInsert e IUpdate são também criadas com os métodos necessários para o administrador inserir e atualizar os campos a que tem acesso.

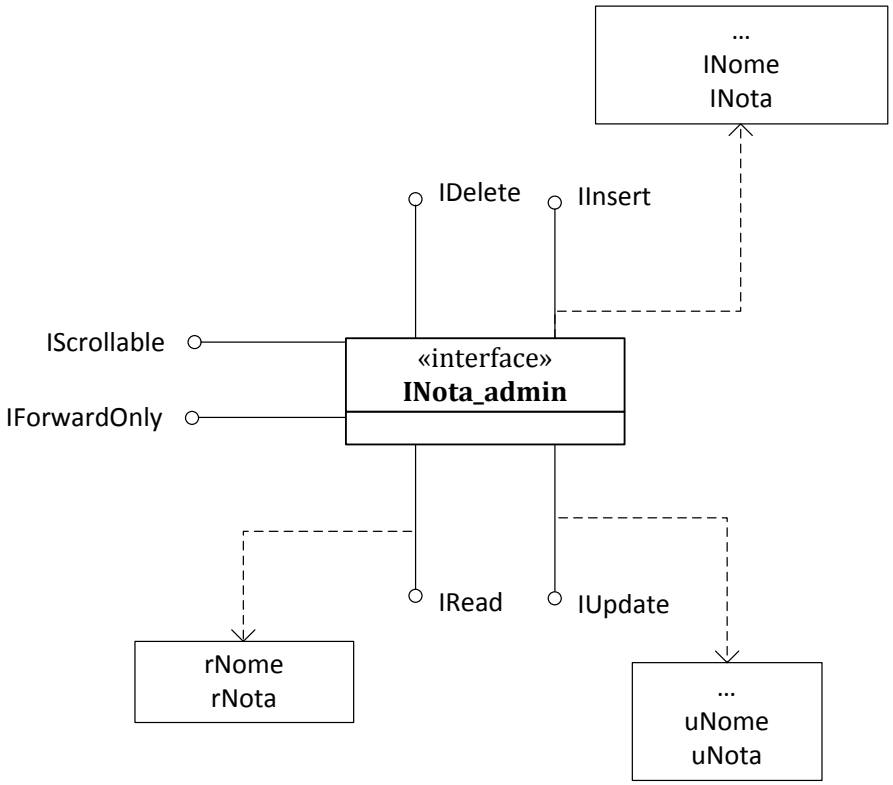


Figura 52 - Interface ELM relativa ao administrador

No caso do Professor o esquema ELM desenvolvido encontra-se na Figura 53.

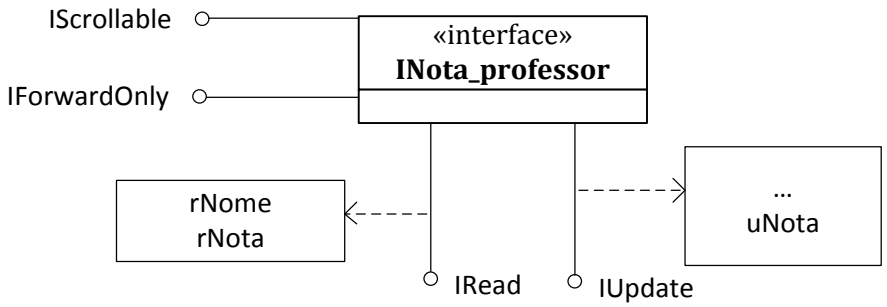


Figura 53 - Interface ELM relativa ao professor

Nesta figura podemos verificar a não existência da interface IDelete e IInsert, visto que o Professor não tem nem permissão para apagar linhas, nem permissão para inserção de novas linhas. Na interface IUpdate apenas se atribui permissão para atualização do campo Nota, sendo retirado o método uNome.

No caso do Aluno o esquema ELM desenvolvido encontra-se na Figura 54.

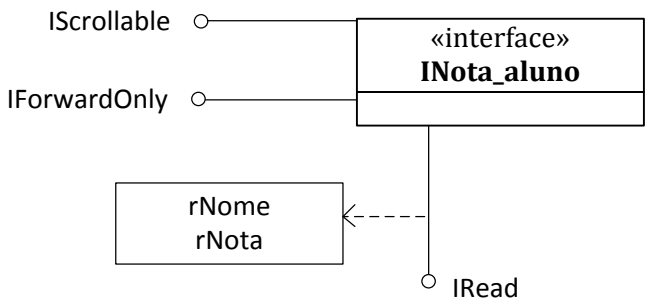


Figura 54 - Interface ELM relativa ao aluno

Neste caso apenas se atribuiu permissão de leitura para os dados da tabela conforme as especificações.

A construção deste conjunto de Esquemas de Negócio possibilita a especificação de políticas internas para permitir ou negar o acesso a campos de tabelas da base de dados. A lógica de negócio é um conjunto destes esquemas de negócio. O desenvolvimento de aplicações é facilitado, com a utilização de ferramentas de desenvolvimento como netbeans ou visualstudio, devido à visualização dos métodos a que temos acesso durante a fase de desenvolvimento. Na Figura 55 e Figura 56 encontramos a informação sobre os métodos acessíveis ao administrador e aluno respetivamente.

Figura 55- Métodos acessíveis ao administrador

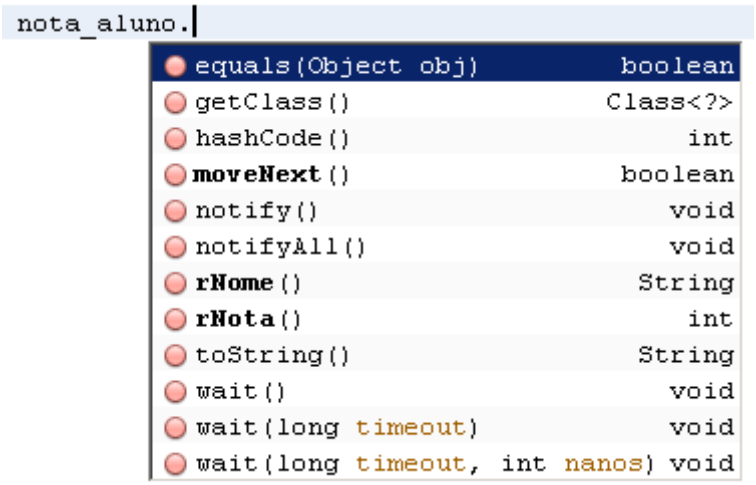


Figura 56 - Métodos acessíveis ao aluno

Exemplos de utilização dos Esquemas de Negócio

Para exemplo de utilização destes Esquemas de Negócio, considere que temos permissão no servidor de políticas para o Esquema de Negócio desenvolvido "nota_admin" e que foi corretamente instanciado pelo gestor de negócios com a seguinte expressão CRUD:

"Select * From Notas"

Esta expressão permite visualizar todos os campos da tabela notas. Na Figura 57 encontramos o código utilizado para o caso do administrador querer imprimir uma listagem com as notas de todos os alunos e os respetivos nomes.

```
nota_admin.execute();
while(nota_admin.moveNext())
{
    System.out.println(nota_admin.rNome()+" "+nota_admin.rNota());
}
```

Figura 57 - Exemplo de impressão de listagem com utilização dos serviços de negócio

Neste caso, executamos o comando execute para execução da expressão CRUD na base de dados relacional correspondente, sendo a ELM preenchida com os dados. Em seguida é utilizado um ciclo while que executa o método moveNext. Este método retorna

um booleano, inferindo se existe uma entrada na próxima linha da tabela. Deste modo percorremos todos os resultados, movendo o cursor para a próxima linha. Para acesso ao valor dos campos utilizamos os métodos existentes na interface IRead, rNome e rNota, para obtenção de nome e da nota respetiva.

Considere agora, o caso em que administrador quer inserir uma nova entrada com o nome e nota. Neste caso é utilizada a interface IInsert, como pode ser visto pelo código da Figura 58.

```
nota_admin.execute();
nota_admin.beginInsert();
nota_admin.iNome("Rui");
nota_admin.iNota(10);
nota_admin.endInsert(true);
```

Figura 58 - Exemplo de inserção de nova linha com utilização dos serviços de negócio

Neste caso, sinalizamos o início de inserção com o método *beginInsert* e de seguida usamos os métodos *iNome* e *iNota* para inserir o nome e a nota nesta nova linha temporária. Com a execução do método *endInsert* é terminada a inserção da nova entrada e as alterações são repercutidas para a base de dados.

Considere que o administrador se enganou na inserção da nota e quer alterara-la. O código exemplo para este caso encontra-se na Figura 59.

```
nota_admin.execute();
while(nota_admin.moveNext())
{
    if(nota_admin.rNome().compareTo("Rui")==0)
    {
        nota_admin.beginUpdate();
        nota_admin.uNota(12);
        nota_admin.updateRow();
        break;
    }
}
```

Figura 59 - Exemplo de atualização de campo com utilização dos serviços de negócio

Inicialmente procedemos à execução do CRUD através do método execute. Em seguida é utilizado um ciclo *while* para encontrar a entrada inserida, comparando o nome

da linha atual da tabela com o nome inserido anteriormente (“Rui”); Quando for encontrada a entrada com o nome correspondente, começamos a atualização da linha com o comando *beginUpdate* e de seguida atualizamos o campo com a utilização do comando *uNota*. Por fim executamos o comando *updateRow* para repercutir as alterações realizadas na base de dados correspondente.

Para eliminar esta linha recentemente inserida podemos utilizar o comando deleteRow, como se pode ver pela Figura 60.

```
nota_admin.execute();
while(nota_admin.moveNext())
{
    if(nota_admin.rNome().compareTo("Rui")==0)
    {
        nota_admin.deleteRow();
        break;
    }
}
```

Figura 60 - Exemplo de remoção de linha com utilização dos serviços de negócio

Conseguimos assim desenvolver um sistema fácil de trabalhar e eficaz, onde se aplicam políticas de acesso obrigatórias internas, através da extensão de interfaces e da criação de métodos especializados nestas.

3.3.2 Gestor de Negócios

O gestor de negócios é o componente que permite:

- Obtenção de conexão à base de dados para os serviços de negócio operarem e a utilização de transações para um maior controlo das operações na base de dados;
- Construção em tempo real da lógica de negócio;
- Carregamento dinâmico em tempo de execução dos serviços de negócio;
- Aplicação das políticas de acesso aos utilizadores.

Uma introdução à criação de serviços de negócio é apresentada no paper [Brant, '00]. Este apresenta a criação dinâmica de relatórios, que mapeiam as tabelas da base de dados relacionais em objetos funcionais que podem ser utilizados pelos utilizadores. Inicialmente no nosso caso, tal como no paper, os esquemas de negócio eram predefinidos antes da execução da aplicação. Isto permite a existência de menos bugs e erros. Este tipo de solução não é dinâmica o suficiente e por conseguinte, teríamos que criar um sistema

flexível e configurável pelo utilizador. Neste paper são também introduzidos os objetos query, cujo principal objetivo é a exibição da informação resultante do query executado, mas existia um problema nesta perspetiva. Consideremos o caso em que tínhamos uma tabela com as vendas realizadas para vários países e queríamos a seleção separada de informações sobre vendas, para Portugal ou Espanha.

Neste caso tínhamos duas soluções:

- Construção de dois Esquemas de Negócio para cada venda realizada;
- Criação de um Esquema de Negócio, adaptável às duas vendas realizadas.

A primeira solução não era fazível para o caso de existência de numerosas vendas. Dai a nossa solução foi o carregamento dinâmico de diferentes expressões CRUD. Neste caso, definiríamos no servidor de políticas duas expressões CRUD, correspondentes a vendas para Portugal e Espanha e na instanciação do serviço de negócio era escolhido qual a expressão CRUD a utilizar. Assim o utilizador cria um esquema de negócio com os campos a que pretendia aceder como se pode verificar no ponto 3.3.1.1, colocando os campos referentes às duas tabelas. Com a criação deste tipo de interfaces, o utilizador pode de seguida, realizar o carregamento deste esquema de negócio através do gestor de negócio, escolhendo qual a expressão CRUD que quer executar.

Com a criação dinâmica em tempo de execução da lógica de negócio de acordo das políticas e o carregamento dinâmico da expressão CRUD, temos um sistema flexível e capaz de se adaptar dinamicamente às políticas de controlo de acesso. Após definição das necessidades para responder ao problema proposto, chegamos ao seguinte esquema geral, que se encontra na Figura 61. Neste esquema geral, podemos verificar a existência de uma classe principal, designada de *Manager*, que permite o acesso a todas as outras interfaces. Esta contém o método *getInstance*, para instanciação do gestor de negócios. Este método recebe cinco parâmetros, sendo os primeiros três referentes aos dados para *login* do utilizador na base de dados relacional, constituídos por:

- un: Correspondente ao *username* do utilizador para *login* na base de dados relacional onde serão executadas as expressões CRUD;
- password: Correspondente à *password* do utilizador para *login* na base de dados relacional;
- urlDB: Corresponde ao url para ligação à base de dados onde são executadas as expressões CRUD.

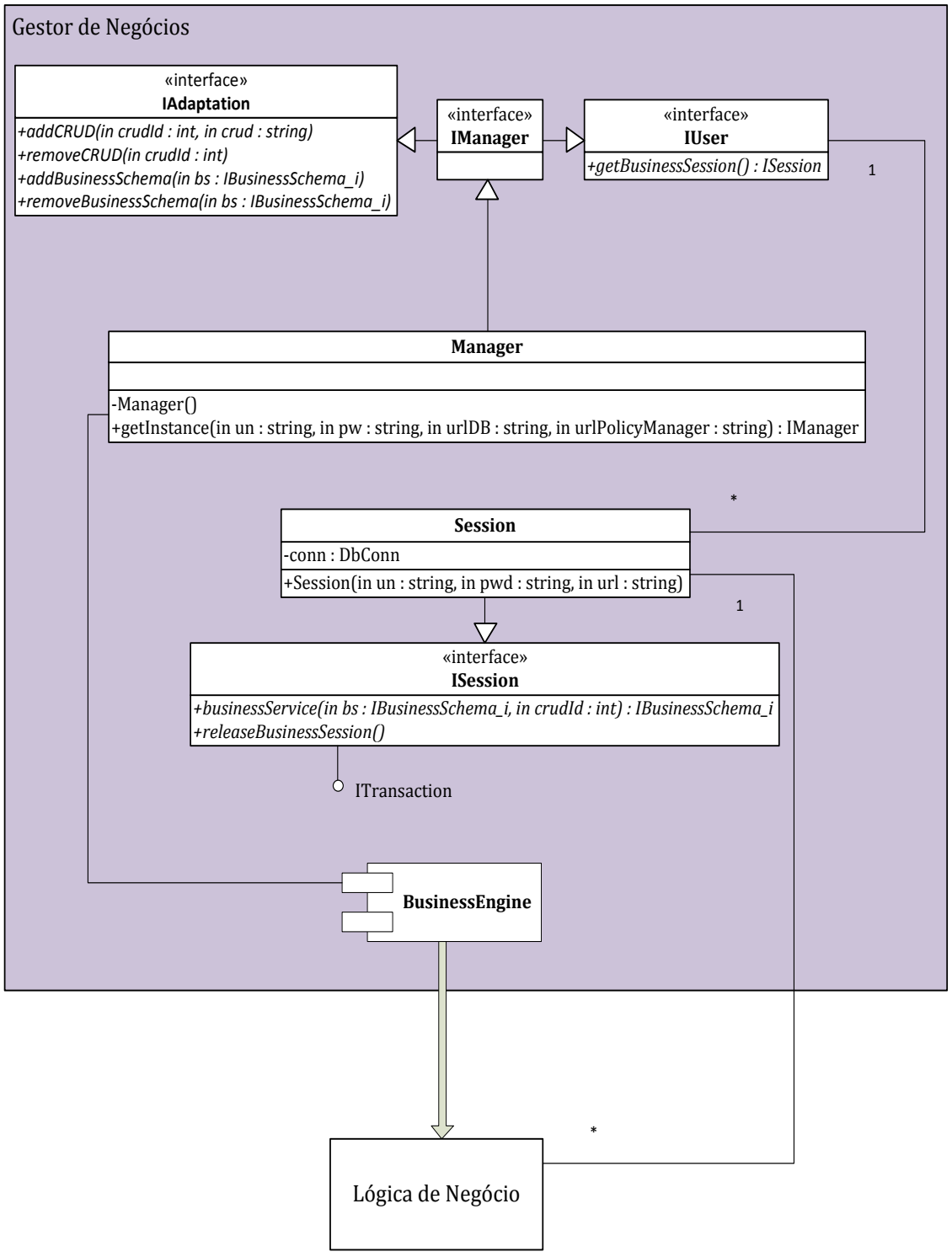


Figura 61 - Esquema geral do Gestor de Negócios

O próximo parâmetro corresponde ao url de configuração para ligação ao servidor de políticas e este é internamente composto pela seguinte sintaxe:

IP:"Ip do servidor":"Porta do servidor";usr:"username";pwd:"password";app:"App";

Onde configuramos os vários campos separando-os por ponto e vírgula. Exemplo:

"Ip:localhost:9000;usr:usr1;pwd:usr1;app:App"

No campo ip é colocado o ip do gestor de políticas e a porta separados por :. No campo usr e pwd, são colocados o username e a password respetivamente. Por último, no campo App é colocada uma referência para a aplicação a ser utilizada.

Depois da correta instanciação do gestor de negócios, este constrói a lógica de negócio partindo das políticas de controlo de acesso definidas no servidor de políticas. Esta construção é realizada através da interface IAdaptation a qual é escondida do utilizador. Com a correta criação da lógica de negócio, o utilizador procede à obtenção de uma sessão (classe Session) através do método getBusinessSession da interface IUser. Esta sessão fornece uma ligação à base de dados (onde se encontram os dados a serem acedidos pelos serviços de negócio) e permite o carregamento dos serviços de negócio que compõem a lógica de negócio.

Nos próximos parágrafos são apresentados em detalhe, a forma como as sessões e transações são geridas, e o modo como o gestor de negócios constrói e gere a lógica de negócio.

3.3.2.1 Sessões e Transações

Para obter conexão à base de dados relacional utilizamos um conjunto de sessões. Uma sessão representa uma forma interativa de troca de informação, conhecida também como diálogo, conversação ou reunião. Por cada sessão, um utilizador possui uma ligação à base de dados relacional na qual são executados os métodos fornecidos pela lógica de negócio.

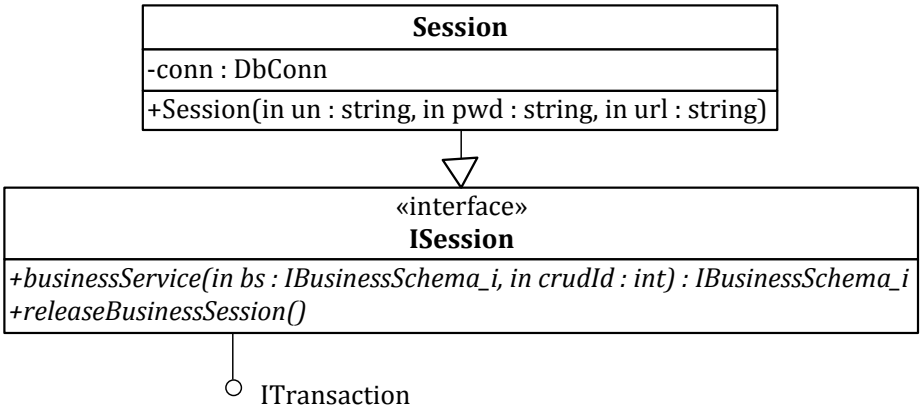


Figura 62 - Modelo do conceito de Sessão Implementado

Para obter uma conexão diferente por cada sessão foi criado o construtor Session (Ver Figura 62). Este recebe como parâmetros: o nome de utilizador, a password e o url referente à localização da base de dados. Com a sua chamada é criada uma conexão para a base de dados relacional, que é armazenada na variável privada Conn. O método businessService (da inteface ISession) permite retornar instâncias dos serviços de negócio. Este método é descrito em detalhe no exemplo mais à frente. Por último, o método releaseBusinessSession, quando executado, liberta todos os recursos (Serviços de negócio) associados à sessão. A manipulação de transações é realizada a partir da interface ITransaction. Esta interface fornece métodos para alterar o nível de isolamento, reverter alterações efetuadas a base de dados e possibilitar a utilização de savepoints. Um utilizador pode estabelecer um savepoint ou um marcador dentro de uma transação. Este marcador define a localização à qual uma transação pode voltar, se parte da transação for cancelada condicionalmente. Os métodos da interface ITransactions estão descritos na Tabela 16.

Por defeito, por cada sessão iniciada, o valor de AutoCommit é verdadeiro. Isto indica que por cada execução de um método dos serviços de negócio, as alterações são imediatamente enviadas para a base de dados relacional. Para utilização das transações temos que executar o método setAutoCommit com parâmetro de entrada, um booleano com valor falso. Deste modo, o envio das alterações é realizado apenas quando necessário.

Métodos	Descrição
`void commit()`	Envia as alterações efetuadas para a base de dados.
`int getTransactionIsolationLevel()`	Obtém o nível de isolamento das transações.
`void releaseSavepoint(Savepoint savepoint)`	Liberta o SavePoint.
`void rollBack()`	Cancela uma transação inteira.
`void rollBack(Savepoint savepoint)`	A transação retorna ao SavePoint especificado e cancela todas as alterações efetuadas na base de dados.
`void setAutoCommit(boolean autoCommit)`	Se autoCommit a verdadeiro, existe sempre um *commit* (envio da informação) a seguir à execução de cada CRUD. Por defeito, todos os CRUD são imediatamente enviados para a base de dados, aquando da sua execução.
`Savepoint setSavepoint()`	Cria um SavePoint e retorna o objeto que o representa.
`Savepoint setSaveSavepoint(String name)`	Cria um SavePoint com um nome específico e retorna o objeto que o representa.
`setTransactionIsolation(int level)`	Especifica o nível de isolamento das transações.

Tabela 16 - Descrição dos métodos utilizados nas operações de transações

Como podemos ver pelo exemplo da Figura 63, colocamos o envio automático de alterações a falso e realizamos commit (envio) das alterações apenas no fim da chamada de todos os comandos necessários, sendo estes enviados juntos como uma unidade singular de trabalho. Caso ocorra um erro no envio das alterações, a exceção é tratada no *catch*, com uma operação de rollback, que reverte as alterações realizadas durante a transação. Deste modo, possuímos um maior controlo sobre o conjunto de alterações que

são efetuadas na base de dados e podemos reverter alterações que poderiam deixar a base de dados num estado inconsistente.

```
        try {
session.setAutoCommit(false);
nota_admin.execute();
while(nota_admin.moveNext())
{
    if(nota_admin.rNome().compareTo("Rui")==0)
    {
        nota_admin.beginUpdate();
        nota_admin.uNota(12);
        nota_admin.updateRow();
        session.commit();
        break;
    }
}
        } catch (SQLException ex) {
    try {
        session.rollBack();
    } catch (SQLException ex1) {
        System.err.println("COULD NOT ROLLBACK TRANSACTION");
    }
}
```

Figura 63 - Exemplo de utilização de transações

Nos próximos parágrafos apresentam-se as estratégias utilizadas para a realização da gestão da lógica de negócio.

3.3.2.2 Gestão da Lógica de Negócio

O gestor de negócios precisa de armazenar informações sobre quais os esquemas de negócio e expressões CRUD existentes. Isto permite ao gestor de negócio realizar a correta gestão da lógica de negócio. Para armazenar as informações sobre os esquemas de negócio e respetivos CRUDs no gestor de negócios foi utilizada a classe HashMap [Oracle, '12b] em Java e a classe HashTable [Microsoft, '12e] em C#. Ambas as classes fornecem conjuntos chave-valor. A diferença chave entre elas é que o acesso numa HashTable é sincronizado mas no nosso caso é irrelevante visto que apenas uma entidade adiciona os elementos ao gestor de negócio. Estas classes permitem a criação de uma estrutura de dados, que utilizam uma função hash para mapeamento dos valores de identificação conhecidos como chaves e os seus valores associados. Para adição de elementos (chave-valor) é utilizado o método put do objeto hashmap em java, ou o método Add da hashtable em C#, como podemos verificar pela Figura 64.

```
JAVA    hashmap.put("chave", 1);
C#      hashtable.Add("chave", 1);
```

Figura 64 - Adição de elementos a HashMap e HashTable em Java e C#

Para remoção de chaves e respetivos valores é utilizado o método *remove* tanto no hashmap em java, como na hashtable em C#, como podemos verificar pela Figura 65.

```
JAVA    hashmap.remove("chave");
C#      hashtable.Remove("chave");
```

Figura 65 - Remoção de elementos do HashMap e da HashTable em Java e C#

A verificação de existência de uma chave é realizada, executando o método *containsKey* em Java e o método contains em C#, como podemos verificar pela Figura 66.

```
JAVA    hashmap.containsKey("chave");
C#      hashtable.Contains("chave");
```

Figura 66 - Verificação de existência de chave no HashMap e HashTable em Java e C#

O armazenamento dos esquemas de negócio e expressões CRUD é realizado, utilizando uma conjunção de dois *HashMaps*/*HashTables* (Ver Figura 67).

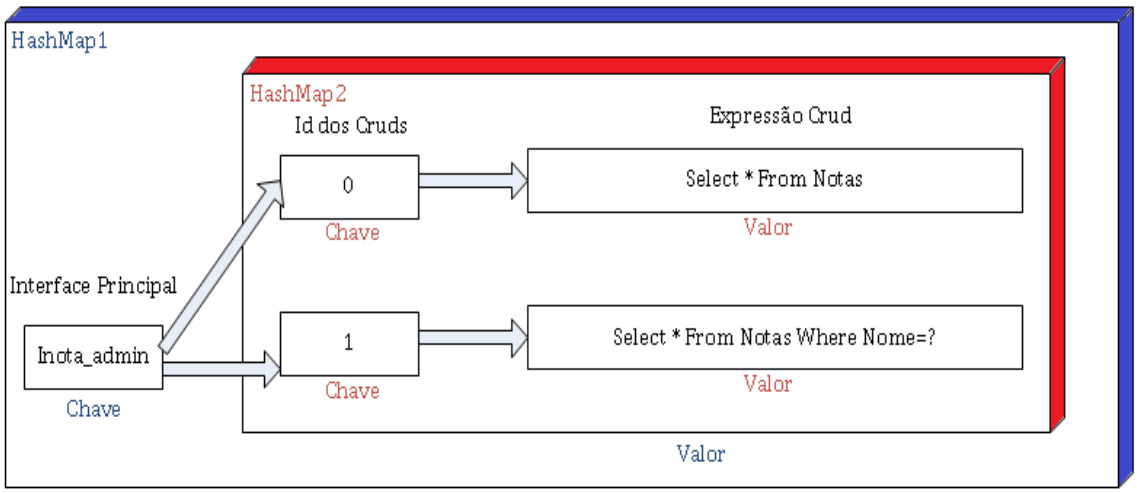


Figura 67 - Esquema principal da solução para armazenamento de informações sobre lógica de negócio

Podemos verificar que a HashMap1 abrange a HashMap2, e nesta são armazenados como chave a classe correspondente à interface principal dos esquemas de negócio e como valor a hashmap2. Na hashmap2 são armazenados como chave o id da expressão CRUD, tal como na base de dados do servidor de políticas, e como valor a string contendo a expressão CRUD. Temos assim uma estrutura que permite um fácil armazenamento da informação necessária ao gestor de políticas. Para a gestão dos esquemas de negócio e dos respetivos cruds, precisamos de um conjunto de métodos que nos permitam realizar operações de adição e remoção de serviços de negócio, tanto da nossa lógica de negócio, como dos hashmaps/hashtables. Estes métodos estão definidos na interface IAdaptation, cuja descrição dos métodos se mostra na Tabela 17 e o respetivo esquema na Figura 68. Estes métodos estão escondidos do utilizador e são apenas utilizados pelo gestor de negócios para gestão dos esquemas de negócio e CRUDS. A construção dinâmica da lógica de negócio é realizada apenas no Java, como foi referido anteriormente. A versão C# é constituída por uma criação estática da lógica de negócio de modo a demonstrar que a implementação dos métodos dos esquemas de negócio pode ser realizada em várias tecnologias.

Método	Descrição
addCrud	Permite a adição ao hashmap/hashtable de uma expressão CRUD. A expressão CRUD é representada pelo seu id (crudid) e pela String contendo a expressão CRUD. A classe bs permite identificar o esquema de negócio e corresponde à interface principal deste.
removeCrud	Permite a remoção do hashmap/hashtable de uma expressão CRUD com id crudId, a um esquema de negócios representado pela classe bs.
addBusinessChema	Permite a adição de um esquema de negócio, a criação da classe que o implementa na lógica de negócio, e a sua entrada ao hashmap/hashtable
removeBusinessSchema	Permite a remoção de um esquema de negócio, tanto do hashmap/hashtable, como da lógica de negócio

Tabela 17 - Descrição dos métodos da interface IAdaptation

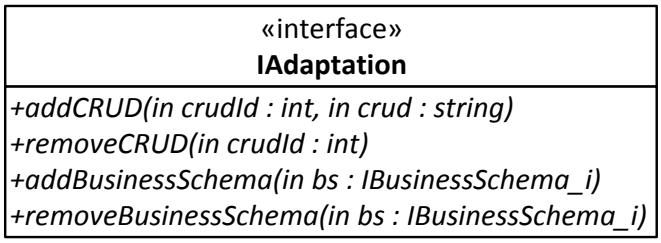


Figura 68 - Esquema da interface IAdaptation

Nos próximos parágrafos vão ser descriminadas as estratégias utilizadas em java para permitir a adição e remoção de esquemas de negócio nos métodos referidos pela interface IAdaptation e a forma como são criadas e carregadas dinamicamente as respetivas implementações em tempo de execução.

3.3.2.2.1 Criação da Lógica de Negócio

Em java, os esquemas de negócio são adicionados e armazenados em conjunto num ficheiro contendo todos os esquemas de negócio, constituindo a lógica de negócio. Para armazenamento em java, foi escolhido o formato de compressão jar (JAVA archive) [Oracle, '12g]. Neste tipo de ficheiros podemos armazenar as interfaces correspondentes aos esquemas de negócio e a sua respetiva implementação. Na Figura 69 estão explicitados os passos para o armazenamento em arquivos java.

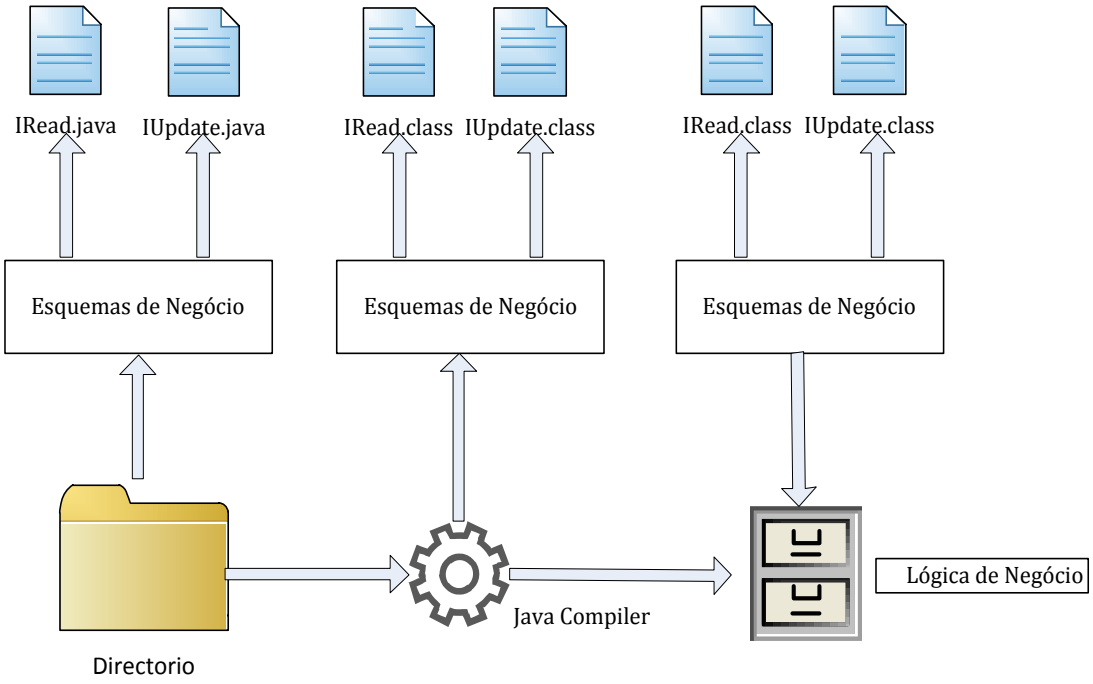


Figura 69 - Armazenamento de Esquemas de Negócio em Java

Os ficheiros .java que contêm as interfaces e a classe de implementação dos esquemas de negócio, são compilados através da ferramenta javaCompiler [Oracle, '12h], sendo depois armazenados em ficheiros .jar.

A utilização do tipo de formato JAR oferece as seguintes vantagens [Oracle, '12g]:

- Segurança: Podemos assinar digitalmente os conteúdos de um ficheiro jar;
- Compressão: O formato JAR permite a compressão dos ficheiros para um armazenamento eficiente;
- Possibilidade de inserção de informações no pacote JAR, tais como fornecedor e versão;
- Portabilidade: O mecanismo para manipulação de ficheiros jar é uma parte *standard* da plataforma Java.

A leitura e a escrita em ficheiros jar são realizadas com a utilização de duas bibliotecas presentes no java:

- java.util.jar.JarInputStream [Oracle, '12d]: permite a leitura de ficheiros jar;
- java.util.jar.JarOutputStream [Oracle, '12e]: permite a escrita de ficheiros jar.

Com a utilização destas *streams* em conjunto com os objetos JarEntry (fornecido pela biblioteca java.util.jar.JarEntry [Oracle, '12c], que representam uma entrada num ficheiro jar), conseguimos gerir os ficheiros jar de uma maneira eficiente e fácil. Nos próximos parágrafos expõe-se a forma como são armazenados os ficheiros neste tipo de arquivos.

3.3.2.2.1.1 Criação de Ficheiros JAR

Para criação de ficheiros jar utilizamos a classe JarOutputStream em conjunto com a classe FileOutputStream, como podemos verificar pela Figura 70.

```
Manifest manifest = new Manifest();
manifest.getMainAttributes().put(Attributes.Name.MANIFEST_VERSION, "1.0");
FileOutputStream stream = new FileOutputStream(tempjar);
JarOutputStream outjar = new JarOutputStream(stream, manifest);
```

Figura 70 - Criação de arquivo JAR

Inicialmente criamos um ficheiro manifesto, que contém informação sobre os ficheiros armazenados (meta-informação). Para a alteração das definições do manifesto,

utiliza-se o comando *getMainAtributes().put*. No nosso caso, queremos criar apenas um ficheiro manifesto padrão que contenha apenas a versão do manifesto (neste caso 1.0) de modo a tornar o ficheiro jar válido. Em seguida, é obtida uma *stream* para o ficheiro a ser criado, utilizando o método FileOutPutStream. Este método recebe como parâmetro um objeto *String* representando o caminho para o ficheiro, que no nosso caso é armazenado na variável tempjar. Por último, é criada uma stream relativa a arquivos jar, através do método construtor JarOutputStream. Este método recebe como parâmetros a nossa *stream* para o ficheiro (variável stream) e o nosso Objeto Manifesto recentemente criado (manifest). Isto permite a criação de um objeto que possibilita a escrita de conteúdos em ficheiros jar (variável *outjar*).

3.3.2.2.1.2 Escrita de Entradas em Ficheiros JAR

Para escrita em ficheiros jar utilizamos a class JarEntry, como podemos verificar na Figura 71. O construtor desta classe recebe como parâmetro, uma String contendo a localização de uma entrada dentro de um arquivo jar. Neste caso encontramos um exemplo para a criação de um ficheiro dentro de um diretório.

```
JarEntry newentryfile = new JarEntry("BusinessInterfaces/file");
outjar.putNextEntry(newentryfile);
outjar.write(buffer, 0, nbytes);
outjar.close();
```

Figura 71 - Exemplo de escrita de um ficheiro num arquivo JAR

O nome do diretório (“BusinessInterfaces”) e o nome do ficheiro (“file”) são especificados como localização da nova entrada. Esta nova entrada é adicionada ao arquivo jar com utilização do método *putNextEntry* do objeto *JarOutputStream*. O objeto JarEntry contém apenas informação acerca de entradas e não o conteúdo das mesmas. Deste modo para a escrita do conteúdo do ficheiro utilizamos o método *write* do objeto *JarOutPutStream*.

Com a utilização destas bibliotecas conseguimos um armazenamento eficaz das interfaces dos esquemas de negócio e da sua respetiva implementação.

3.3.2.2.2 Construção da Implementação dos Esquemas de Negócio

As estratégias utilizadas para a construção dos métodos dos esquemas de negócio foram evidenciadas no ponto 3.3.1.1.2. Para obtenção de informações sobre as várias interfaces que compõem os esquemas de negócio, foi criada uma classe que guarda as várias interfaces e retorna as informações necessárias sobre os métodos para a construção da classe de implementação. Esta classe tem como nome "contexto de entidades de negócio" e utiliza a API Reflection do java para obtenção de informações.

Obtenção e separação das interfaces

Para obter e separar as várias interfaces que compõem os esquemas de negócio foi desenvolvido um método, o qual recebe como parâmetro a interface principal dos esquemas de negócio. Este armazena as interfaces secundárias que a interface principal estende, através da utilização do método *getInterfaces*() da API *Reflection*. Estas interfaces são depois armazenadas num array de objetos do tipo class:

```
Class[] itfs = itf.getInterfaces();
```

A separação das interfaces é realizada com o recurso ao método *getName*, o qual nos permite obter o nome das interfaces. Com o correto armazenamento e separação das interfaces, procedemos à criação da classe de implementação.

Criação da classe de implementação

A implementação dos métodos que compõem a classe de implementação dos esquemas de negócio (tal como especificado no ponto 3.3.1) é realizada a partir da informação sobre os métodos que constituem os esquemas de negócio. Os métodos são obtidos para um array de objetos do tipo Method com a utilização do comando *getMethods*:

```
Method[] tmp = this.itfUpdate.getMethods();
```

O array tmp é de seguida analisado utilizando para tal, os métodos descritos na Tabela 18.

Método	Descrição
getName()	Retorna uma *String* correspondente ao nome do método
getParameterTypes()	Retorna um *array* com os tipos dos parâmetros do método
getReturnType()	Retorna o tipo de retorno do método

Tabela 18 - Descrição dos métodos da classe *Method*

Com a utilização destes métodos é facilmente obtida a informação para construção da classe de implementação.

Compilação

Depois da construção da classe de implementação é necessário realizar a sua compilação. Para tal utiliza-se a interface *JavaCompiler*[Oracle, '12h] da biblioteca javax.tools.JavaCompiler. Esta interface permite a invocação dos compiladores da linguagem de programação Java em tempo de execução. Deste modo podemos compilar os esquemas de negócio em tempo de execução, procedendo-se em seguida ao seu armazenamento num ficheiro jar, criando-se assim a respetiva lógica de negócio.

3.3.2.2.3 Carregamento Dinâmico dos Serviços de Negócio

Com os esquemas de negócio e a respetiva Implementação compilados e armazenados num arquivo jar, podemos realizar o carregamento destes de modo a obter os serviços de negócio. Tal como vimos no Esquema geral do gestor de negócios (ponto 3.3.2), o carregamento dos serviços de negócio é realizado numa Sessão através do método BusinessService. Este método carrega em memória a classe da implementação dos esquemas de negócio, e os seus recursos. Como retorno, este método retorna um objeto genérico <T>, para ir de encontro aos diferentes tipos de interfaces que podem compor os Esquemas de Negócio. Em java foi utilizada a Dynamic proxy API [Blosser, '00] para execução dos métodos da classe de implementação. Uma *proxy* é uma classe que funciona como uma interface para outro objeto, sendo esta designada por dinâmica, pois implementa uma interface em tempo de execução. A utilização de uma *proxy* obriga a que chamadas a métodos ocorram indiretamente, através do objeto *proxy*.

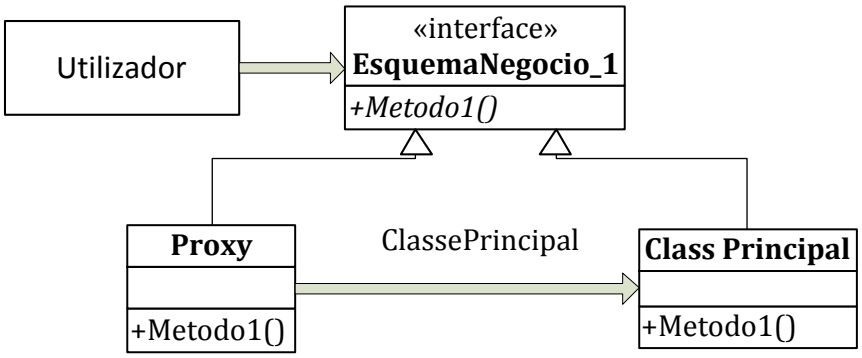


Figura 72 - Esquema de Funcionamento de *proxy* dinâmica

Na Figura 72, podemos verificar que o utilizador tem acesso a um esquema de negócio e que cada chamada a um método desta interface (método1) é realizada na *proxy*. A *proxy* por sua vez, delega a chamada deste método à classe principal que implementa aquele esquema de negócio. Deste modo conseguimos executar os métodos da classe principal de implementação que se encontra na lógica de negócio (ficheiro jar). Nos próximos parágrafos explicamos em detalhe, como se realiza o carregamento da classe de implementação e a sua atribuição a uma *proxy* dinâmica. O carregamento dinâmico das classes que se encontram no jar é realizado utilizando a classe *URLClassLoader* [Oracle, '12p] em conjunto com o método *loadclass* da classe *ClassLoader* [Oracle, '12a].

```
URL url = jarIn.toURI().toURL();

URL[] urls = new URL[]{url};

ClassLoader cl = new URLClassLoader(urls);

BTE = (Class<T>) cl.loadClass(classpath);
```

Figura 73 - Exemplo de Carregamento Dinâmico de classe dentro arquivo JAR

Na Figura 73 encontra-se um exemplo de como efetuar o carregamento de uma classe que se encontra dentro de um arquivo jar (Variável *jarIn*). Inicialmente, obtém-se os urls (localização) de todas as classes e recursos no ficheiro a partir da classe URLClassLoader. Em seguida é efetuado o carregamento da classe através do método loadClass da classe ClassLoader, sendo a classe requerida (identificada através da localização na variável *Classpath*) carregada, para a classe genérica BTE (Class<T> BTE).

Já obtivemos o carregamento dinâmico da nossa classe, agora pretendemos a instanciação dinâmica em tempo de execução desta classe. Na Figura 74 efetuamos o

carregamento do construtor da classe de implementação. Inicialmente é criado uma array de classes (Variável *types*), onde são definidos os tipos de objetos pela ordem exata. De modo a encontrar automaticamente o construtor pretendido, utilizamos o método *getConstructor* da classe genérica BTE, ao qual passamos o array de classes criado anteriormente.

.

```
Class types[] = new Class[2];
types[0] = Connection.class;
types[1] = String.class;
Constructor c = BTE.getConstructor(types);
```

Figura 74 - Carregamento de um construtor em tempo de execução de uma classe genérica

Na Figura 75 encontramos a forma como a classe é carregada para a *proxy* dinâmica. A instanciação da classe genérica é efetuada com o método *newInstance*. Este método pertence ao construtor obtido no passo anterior e recebe como parâmetros de entrada, um array de objetos (correspondentes aos parâmetros a serem atribuídos ao construtor).

```
Object arglist[] = new Object[2];
arglist[0] = conn;
arglist[1] = crud;
return (T) DebugProxy.newInstance(c.newInstance(arglist));
```

Figura 75 - Instanciação de uma classe genérica em tempo de execução

3.3.2.3 Exemplo de Utilização do Gestor de Negócios

Neste capítulo mostramos um exemplo de como utilizar o gestor de negócios para aceder aos serviços que temos acesso. Para utilização do gesto de negócios, é necessário realizar os seguintes passos:

- Obtenção de uma instância do gestor de negócios;
- Obtenção de uma sessão;
- Instanciação dos serviços de negócio a que se tem acesso.

Obtenção de uma instância do gestor de negócios

Para obtenção de uma instância do gestor de negócios utilizamos o método *getInstance* da *class* principal *Manager*, descrita em detalhe no ponto 3.3.2.

Exemplo:

```
IManager gestor = Manager.getInstance("username", "password", url, serverurl,true);
```

Deste modo podemos obter uma instância do gestor de negócios.

Obtenção de uma sessão

A sessão é obtida através do método *getBusinessSession*, presente na interface IUser.

Exemplo:

```
ISession sessao=gestor.getBusinessSession();
```

Neste caso, a nossa sessão é obtida através do objeto gestor instanciado no passo anterior. Com a correta criação da sessão podemos de seguida realizar a instanciação dos serviços de negócio aos quais temos direito.

Instanciação dos serviços de negócio

A instanciação dos serviços de negócio é realizada através da chamada ao método *businessService*, presente na interface ISession. Este método recebe como parâmetros a interface principal do esquema do negócio e o id correspondente ao CRUD a ser executado. Para exemplo considere que tínhamos acesso ao esquema de negócios ICat_s:

```
ICat_s selectAllCategories = sessao.businessService(ICat_s.class, crudid);
```

Neste caso, carregamos o nosso serviço de negócio, dinamicamente em tempo de execução para o esquema de negócio *ICat_s* e com o *id* respetivo da expressão CRUD requerida, *crudid*.

3.3.3 Servidor de Políticas

Neste capítulo são mostrados os passos necessários para a criação do nosso servidor de políticas, que armazena as políticas de controlo de acesso estabelecidas. Vamos estabelecer o desenvolvimento do servidor de políticas em três passos:

- Definição das políticas de segurança pelas quais o nosso sistema é regulado;
- Definição dos modelos conceptual e lógico, que nos permitam armazenar as várias políticas de acesso em base de dados relacionais;
- Definição dos algoritmos que permitam estabelecer os mecanismos de segurança para estabelecimento das políticas de acesso.

3.3.3.1 Políticas de Segurança

O servidor de políticas desenvolvido deve permitir:

- Armazenar informação sobre utilizadores e sessões;
- Armazenar informações sobre as Aplicações;
- Armazenar Esquemas de negócio e os CRUDS respetivos;
- Utilização do controlo de acesso RBAC (Descrito no ponto 2.1.3);
- Permitir que cada cliente seja associado a uma aplicação e que consoante a aplicação, tenha acesso a um conjunto de papéis. A estes papéis são associados um conjunto de Esquemas de Negocio, que por sua vez, permitem acesso a um conjunto de expressões CRUD.

Nos próximos parágrafos descrevem-se os modelos desenvolvidos para permitir atingir os objetivos descritos neste ponto.

3.3.3.2 Modelos de Segurança

Neste capítulo são apresentadas as soluções encontradas para resolução do problema proposto, onde iremos apresentar os vários modelos desenvolvidos (Modelo Conceptual e Lógico), que permitem o armazenamento das nossas políticas de controlo de acesso.

3.3.3.2.1 Definição do Modelo Conceptual

Para armazenamento das políticas de acesso foi utilizado o controlo de acesso baseado em papéis, pois permite um maior controlo sobre as políticas e é facilmente implementado numa base de dados relacional. Neste modelo conceptual pretende-se obter já um modelo que permita atingir os objetivos especificados no ponto 3.3.3.1. Utilizamos

para tal o modelo RBAC1 (descrito em 2.1.3) que permite a existência de, sessões, autorizações, utilizadores e um conjunto de papéis numa estrutura do tipo hierárquica. A obtenção de uma estrutura de papéis hierárquica é realizada através da adição de informação a um papel, sobre o próximo papel na hierarquia. No nosso caso, é guardada a referência para o papel hierarquicamente superior, designado de papel pai. A utilização de uma estrutura hierárquica de papéis é importante, visto que simplifica a gestão de papéis.

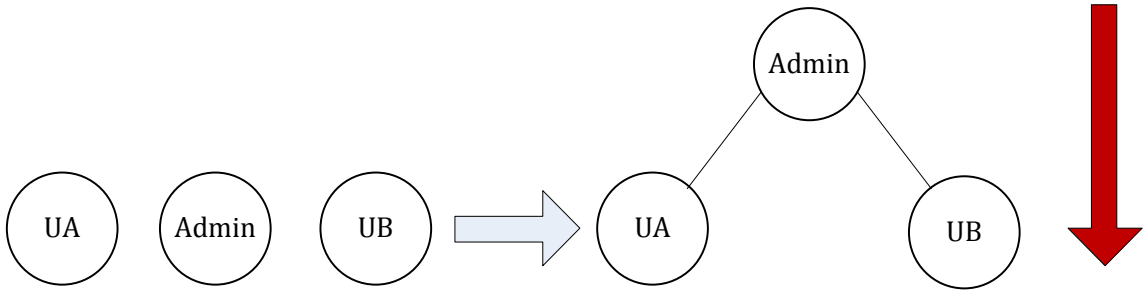


Figura 76 - Exemplo prático de propagação de papéis

Tomemos como exemplo a Figura 76, onde existem três papéis: UA, Admin e UB. Considere que queremos atribuir a autorização ao papel Admin. Além das suas autorizações inerentes também são necessárias as autorizações do papel UA e papel UB. No caso de a estrutura de papéis ser uma estrutura livre, sem qualquer tipo de hierarquia, era necessário atribuir a este utilizador a autorização para aceder aos papéis Admin, UA e UB, sendo necessárias três autorizações para atingir o objetivo proposto. No caso de uma estrutura de papéis organizada numa hierarquia, define-se os papéis UA e UB, como sendo papéis filhos do papel Admin, permitindo assim a construção de uma hierarquia, como se pode verificar pelo esquema na parte direita da figura. Neste caso, é apenas atribuída a autorização ao papel Admin, existindo uma propagação das permissões, representada pela seta a vermelho. O modelo RBAC desenvolvido permite a atribuição de autorizações e de delegações, sendo que as delegações são atribuídas com fins temporários e as autorizações de uma forma persistente. Chegamos assim ao seguinte modelo conceptual, presente na Figura 77 e no qual se indica o nome das entidades e a relação entre estas.

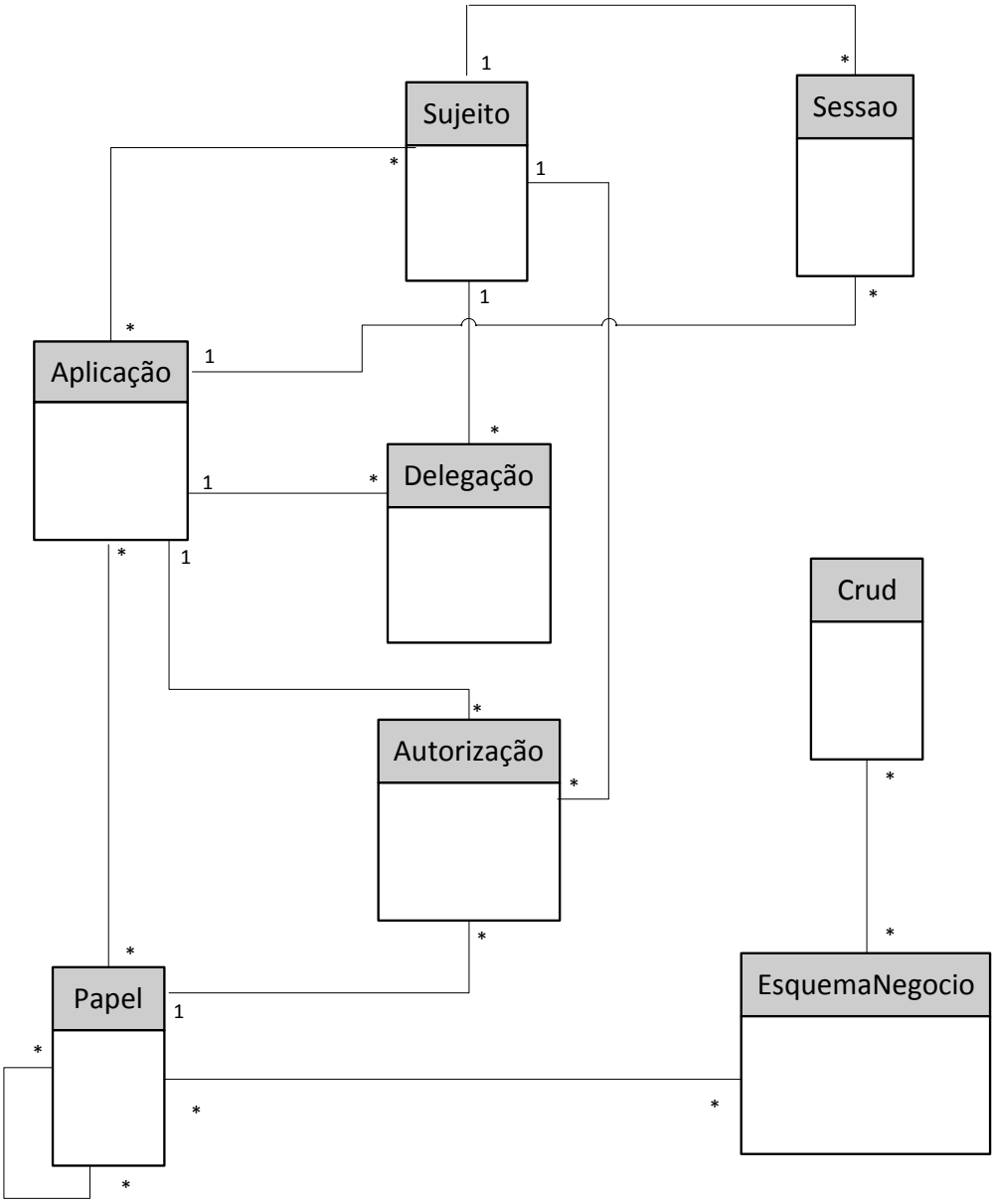


Figura 77 - Modelo Conceptual do Servidor de Políticas

Neste modelo podemos encontrar já a nossa definição de políticas de acesso, através da existência de utilizadores (Tabela Sujeito) e Aplicações (Tabela Aplicação), e da atribuição de autorizações (Tabela Autorização) e delegações (Tabela Delegação) a papéis. Podemos verificar que para cada Aplicação existe um conjunto de sujeitos associados e um conjunto de papéis associados. A nossa entidade *papel* inclui uma ligação para si mesma. Esta ligação traduz-se numa composição hierárquica, em que a atribuição de permissões a papéis de nível superior propaga-se para os papéis filhos destas. Aos papéis são associados um conjunto de esquemas de negócio, que por sua vez possuem um conjunto

de expressões CRUD associadas. Assim ao atribuirmos permissão a papéis, estamos a permitir o acesso a determinados esquemas de negócio e aos seus respetivos CRUDS. A tabela de gestão de sessões (Tabela Sessao) é utilizada para guardar a informação relativa a Aplicações e Sujeitos ativos no sistema de modo a serem reportadas as alterações efetuadas às políticas.

3.3.3.2.2 Definição do Modelo Lógico

Para construção do modelo lógico procedemos à introdução dos atributos nas tabelas e à criação das tabelas intermédias. O nome das tabelas foi normalizado, seguindo a utilização de três letras para criar referências a tabelas e nas tabelas intermédias a utilização da conjunção das três letras de cada tabela utilizada. Chegamos assim ao esquema lógico relativo ao servidor de políticas representado na Figura 78. Podemos verificar a existência de um campo designado por "*ref* " nas tabelas Aplicação, Papel, EsquemaNegócio e Crud. Este campo foi criado para possibilitar a identificação destas entidades ao nível de programação. A tabela autorização contém um campo (*Codigo*) para identificação de um código de autorização. Este código permite a criação de uma relação com outra tabela, onde definimos mensagens com informações adicionais sobre esta autorização. A estrutura hierárquica de papéis é obtida com a introdução de um campo "RolRol_id" que permite armazenar o *id* referente a outro elemento da tabela Papeis. No nosso caso, este *id* é referente ao papel pai do papel onde este se encontra. Os papéis raiz, que são os papéis que se encontram no topo da rede hierárquica não possuem um papel pai. Deste modo a solução encontrada passou pela possibilidade de inserir valores do tipo Null neste campo. A disponibilização das interfaces correspondentes aos esquemas de negócio é necessária para a utilização dos serviços de negócio. Deste modo foi criado o campo App_Esquemas_Negócio que possibilita o armazenamento de um ficheiro do tipo .jar com as interfaces dos esquemas de negócio daquela Aplicação. Isto possibilita por exemplo, o carregamento na ferramenta NetBeans do ficheiro jar de modo a ser possível utilizar os métodos das interfaces para facilitar o desenvolvimento de aplicações.

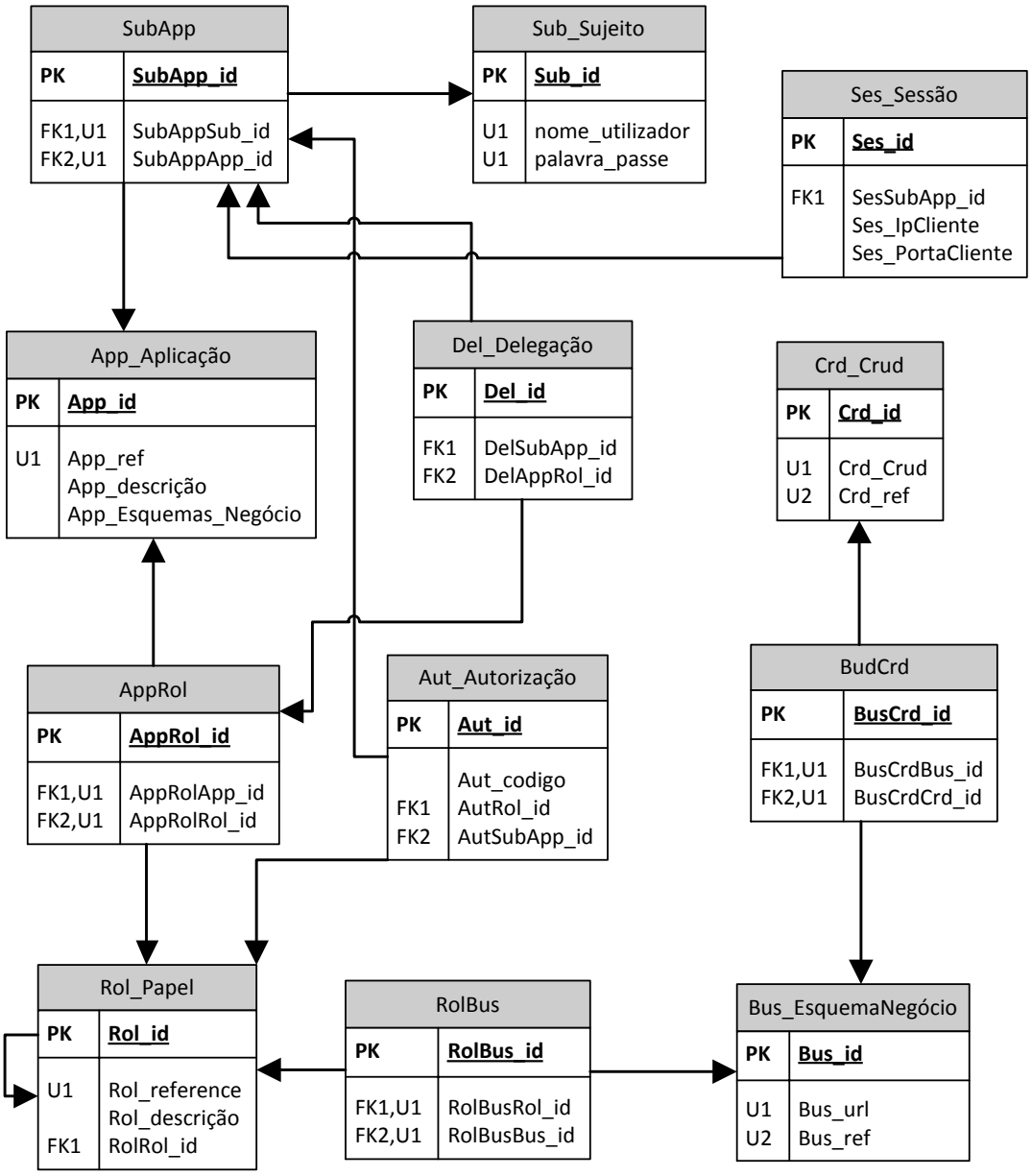


Figura 78 - Modelo Lógico do Servidor de Políticas

Em seguida, descrevem-se as tabelas mais importantes deste esquema:

- Sub_Sujeito: Nesta tabela são armazenadas informações de acesso dos utilizadores. É armazenado o nome do utilizador e a respetiva palavra passe. Esta tabela é necessária para identificação do utilizador no sistema de modo a poderem ser atribuídos os esquemas de negócio e as expressões CRUD a que tem direito;
- App_Aplicação: Nesta tabela são armazenadas informações relativas a aplicações:
    - App_ref: Atributo onde é guardada uma String para identificação da aplicação a nível de programação;

- App_Descrição: Atributo onde é armazenado um texto com a descrição da aplicação, de modo a ajudar os utilizadores a entender a que tipo de negócio esta aplicação se aplica;
- App_EsquemasNegócio: Neste Atributo é armazenado o ficheiro .jar que armazena as interfaces dos vários esquemas de negócio de modo a possibilitar o desenvolvimento de aplicações e a utilização dos serviços de negócio.

- Rol_Papeis: Nesta tabela são armazenadas informações sobre os papéis, como a referência a nível de programação e a sua descrição. Como foi utilizada hierarquia de papéis, foi criado outro campo (*RolRol_id*) com a referência para o papel pai;

- Ses_Sessão: São guardadas informações sobre os utilizadores que estão a utilizar o sistema. Esta informação é recebida no gestor de negócios após *login* no gestor de políticas. São também guardadas informações sobre o ip do gestor de negócios e a respetiva porta, para permitir ao gestor de políticas transmitir as alterações realizadas para a máquina correta;

- Bus_EsquemasNegocio: Nesta tabela são guardadas as informações sobre os esquemas de negócio, tais como:
  - Bus_ref: Atributo onde é guardada uma String para identificação do esquema de negócio a nível de programação;
  - Bus_url: Atributo onde é armazenada a localização da interface principal do esquema de negócio. Este url é utilizado para efetuar o carregamento dos esquemas de negócio pelo gestor de negócios de modo a possibilitar a construção da lógica de negócio.

- Crd_Crud: Nesta tabela são guardadas as expressões CRUD e os respetivos ids, que as permite identificar no carregamento dos serviços de negócio pelo gestor de negócios.

Foi criada também a tabela (Ver Figura 79) para armazenamento de informações sobre os vários gestores de políticas (Ip e Porta). Esta informação permite ao monitor de

políticas identificar onde se encontra o gestor de políticas para reportar as alterações que ocorrem nas políticas de acesso.

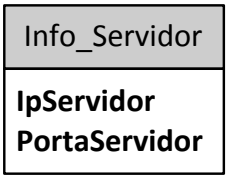


Figura 79 - Tabela com informação sobre o gestor de políticas

3.3.3.3 Mecanismos de Segurança

Neste capítulo são apresentados os algoritmos desenvolvidas para a construção dos mecanismos de segurança que permitem estabelecer a que recursos um utilizador tem acesso.

3.3.3.3.1 Obtenção da Informação Inicial

Inicialmente, aquando do *login* do utilizador no sistema é necessário adquirir todos os papéis a que o utilizador tem acesso. Esta informação é obtida da seguinte maneira:

- Obtenção de todos os papéis autorizados ou delegados;
- Obtenção de todos os papéis filhos dos papéis obtidos anteriormente.

Consideremos como caso prático, o caso representado na Figura 80. Assumindo a existência dos papéis A, B1, B2, C21, C22, D21 e D22. A verde encontram-se os papéis a que o utilizador tem autorização (B2 e C22).

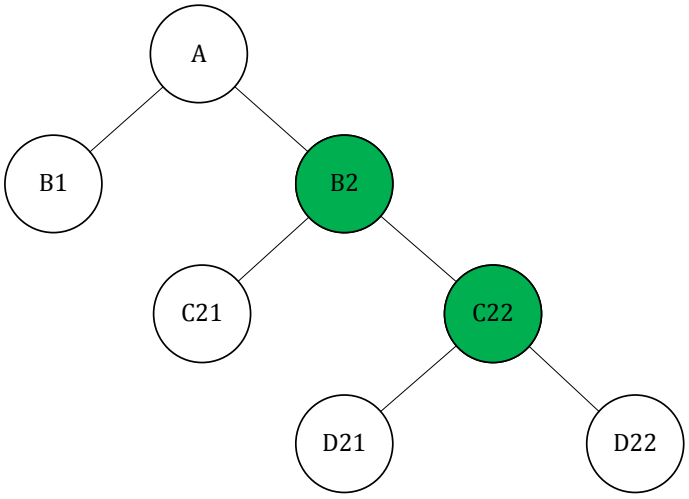


Figura 80 - Esquema de Papéis Exemplo

Para obtenção de todos os papéis a que o utilizador tem direito, foi usado o seguinte algoritmo:

1. Obtenção de todos os papéis autorizados ou delegados, neste caso teríamos os papéis B2 e C22;
2. Pesquisa recursiva pelos papéis filhos destas e que ainda não tenham permissões. Neste caso os papéis pai são B2 e C22, e os papéis filhos destes são C21, D21 e D22. Como estes não possuem papéis filho, a pesquisa recursiva termina.

No fim da pesquisa temos todos os papéis a que os utilizadores têm acesso. Na Figura 81 apresentamos a evolução do algoritmo.

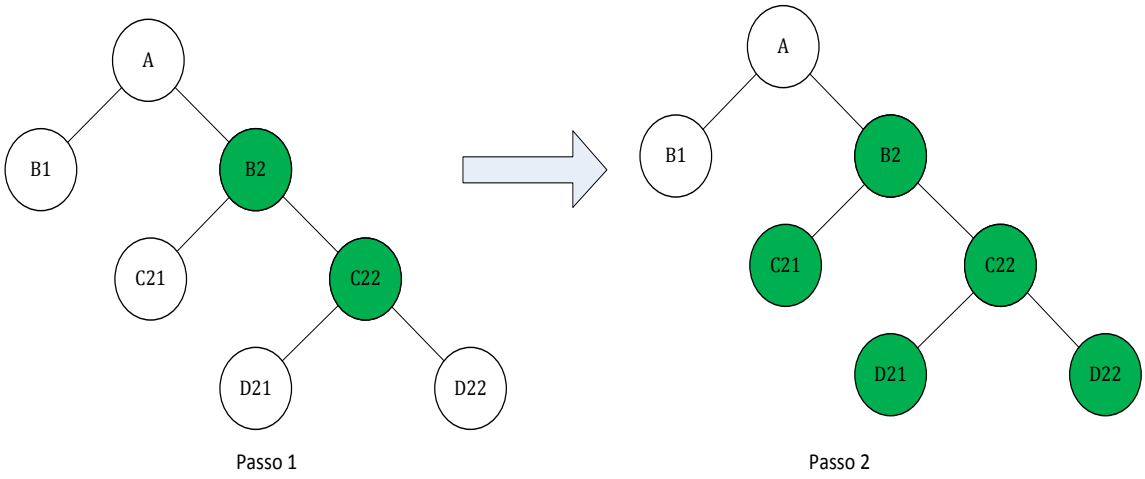


Figura 81 - Exemplo de caso prático para obtenção de informações sobre papéis

Nesta figura os papéis com a cor verde são os papéis para os quais existem permissões (autorizações ou delegações). No passo 1 são obtidos os papéis B2 e C22 através da análise das tabelas de permissões. Em seguida no passo 2, verifica-se os papéis filhos destes recursivamente para os quais não existem permissões. No caso do papel B2, encontramos os papéis filhos C21 e C22, sendo apenas adicionado o papel C21 ao conjunto de papéis a serem enviados, visto que para o papel C22 já existe permissão. Para o caso da pesquisa recursiva do papel C22, são obtidos os papéis D21 e D22.

3.3.3.4Alteração das Políticas

Nos próximos parágrafos apresentamos os algoritmos desenvolvidos para obter as informações necessárias que o gestor de políticas envia ao gestor de negócios quando

ocorrem alterações nas políticas. Numa primeira parte, apresentamos o algoritmo para a adição de autorizações e delegações e numa segunda parte, o algoritmo para remoção destas.

3.3.3.4.1 Adição de Autorizações e Delegações

No caso de adição de autorizações ou delegações, o algoritmo é o seguinte (Considere que os papéis a adicionar são guardados numa lista de papéis):

1. Verificação recursiva até à raiz a partir dos papéis pai se existe já alguma autorização ou delegação atribuída. Se algum papel pai tiver permissão, nenhuma informação é enviada para o cliente. Isto porque, já existe algum tipo de permissão atribuída a este papel a partir de um papel pai e o algoritmo termina. Se não prossegue para passo 2;
2. Realiza a pesquisa recursiva pelos papéis filho que ainda não possuem permissão e adiciona-os à lista de papéis. Quando existir algum papel com permissão, a pesquisa recursiva nesse ramo termina.

No fim teremos todos os novos papéis para os quais existe permissão devido à adição de uma nova autorização ou delegação. Apresentam-se os seguintes casos para exemplo:

Caso1:

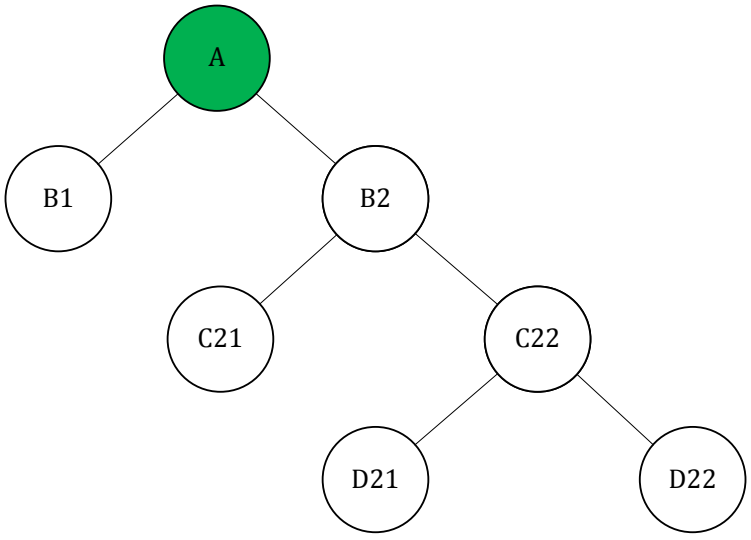


Figura 82 - Exemplo "Prático 1" de adição de Permissões

Considere para este caso a Figura 82, onde existe inicialmente permissão associada ao papel A. Na Figura 83 podemos verificar a evolução do algoritmo no caso onde se atribui permissão ao papel C22. No passo 1 a vermelho, é realizada a pesquisa recursiva pelos papéis pai para verificação de existência de algum papel com permissão. Este passo termina ao encontrar o papel raiz A que já possui permissão, induzindo que o papel C22 também possui permissão, terminando o algoritmo. Deste modo, nenhuma informação é enviada ao utilizador, visto que já está atribuída permissão ao papel C22 através do papel A.

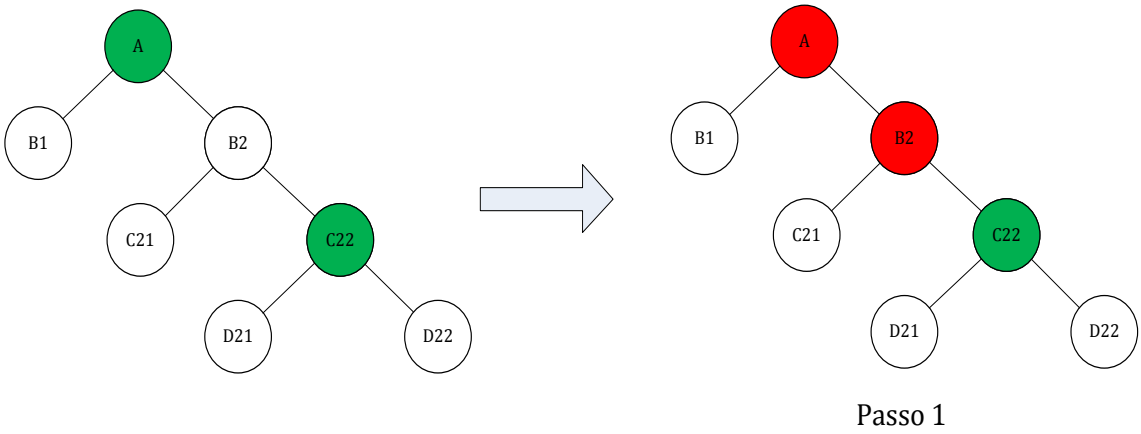


Figura 83 - Evolução do algoritmo para exemplo "Prático 1" com adição de permissões

Caso 2:

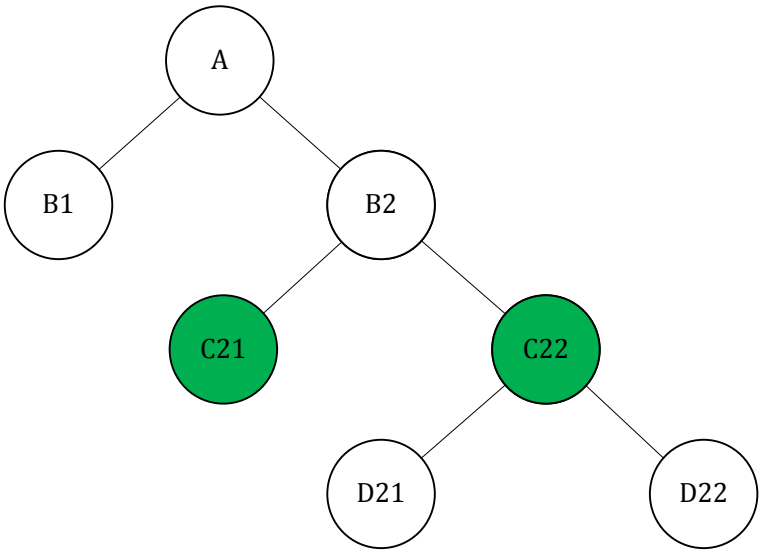


Figura 84 - Exemplo "Prático 2" de adição de permissões

No caso 2 representado na Figura 84 estão atribuídas inicialmente as permissões aos papéis C21 e C22. Como as permissões são atribuídas recursivamente aos papéis filhos destas, então são também atribuídas permissões para os papéis D21 e D22. A evolução do algoritmo depois da adição de permissão ao papel B2 encontra-se na Figura 85.

O passo 1 difere do caso 1, pois neste caso não existe nenhuma permissão atribuída até à raiz a partir dos papéis pai. Prosseguimos então para o passo 2, onde se pesquisa recursivamente os papéis filhos deste. Como os papéis C21 e C22 já se encontram com permissões e não existe mais nenhum ramo da hierarquia para efetuar a pesquisa, apenas o papel B2 é adicionado à lista de papéis a adicionar.

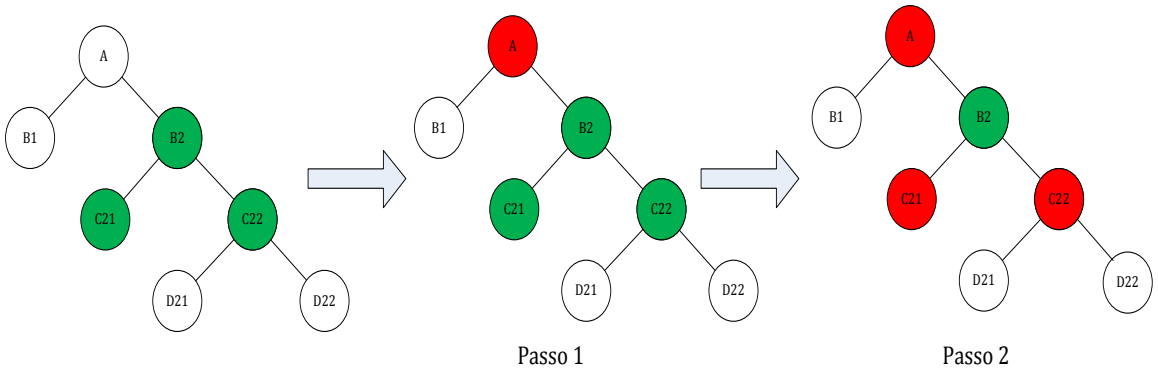


Figura 85 - Exemplo do algoritmo para exemplo "Prático 2" com adição de permissões

3.3.3.4.2 Remoção de Autorizações e Delegações

No caso de remoção de autorizações ou delegações a um papel, o algoritmo é o seguinte (Considere a existência de uma lista com os papéis a serem removidos):

1. Verifica recursivamente até à raiz a partir dos papéis pai se existe já alguma permissão atribuída. Se algum papel tiver permissão, nenhuma informação é enviada para o utilizador. Isto porque ainda existe permissão atribuída a este papel através de um papel pai e algoritmo termina, se não prossegue para passo 2;
2. Adiciona o papel inicial para remover à lista para remoção e verifica recursivamente os papéis filhos que ainda não contenham permissões, sendo estes também adicionadas à lista para remoção.

Apresenta-se o seguinte caso para exemplo que se encontra na Figura 86. Nesta figura podemos verificar que existem duas permissões atribuídas na base de dados: A e C22.

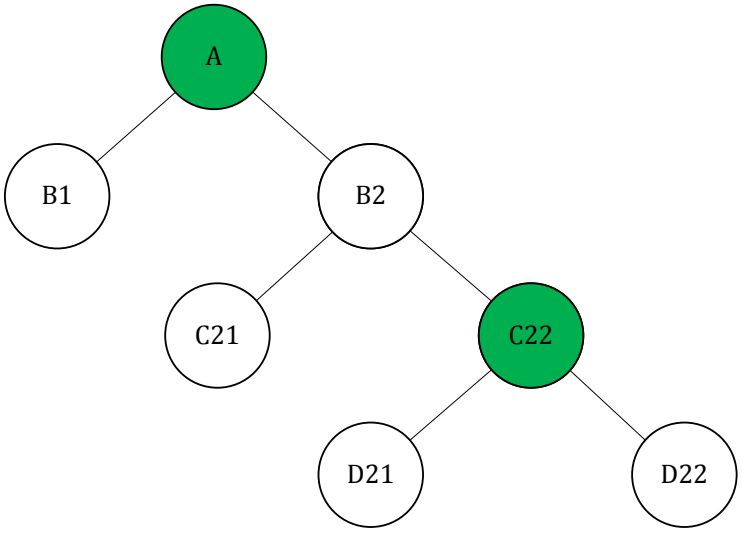


Figura 86 - Exemplo "Prático 1" de remoção de permissões

Considere o caso em que se efetua a remoção da permissão ao papel C22, o algoritmo evolui da maneira exemplificada na Figura 87. Neste caso no passo 1 verifica-se através de uma pesquisa recursiva se existe alguma permissão originada nos papéis pai (a vermelho). Como existe permissão para o papel A, isto implica que também existe permissão para o papel C22, devido à propagação de permissões, terminando o algoritmo.

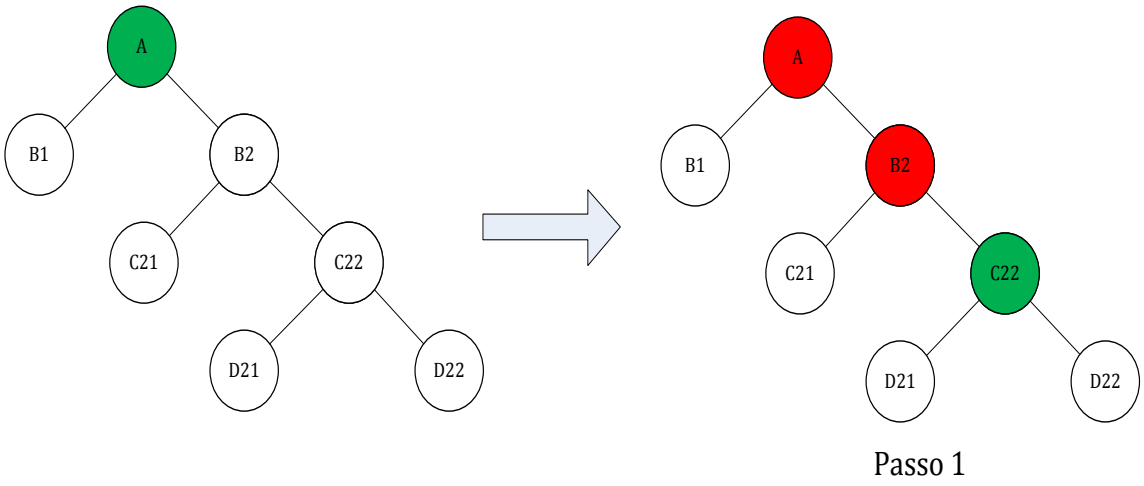


Figura 87 - Evolução do algoritmo para exemplo "Prático 1" no caso de remoção de permissão

Consideremos novamente o caso inicial da Figura 86 com permissões para os papéis A e C22 mas agora, em vez da remoção do papel C22, a remoção do papel A.

A remoção deste papel despoleta a evolução do algoritmo exemplificado pela Figura 88.

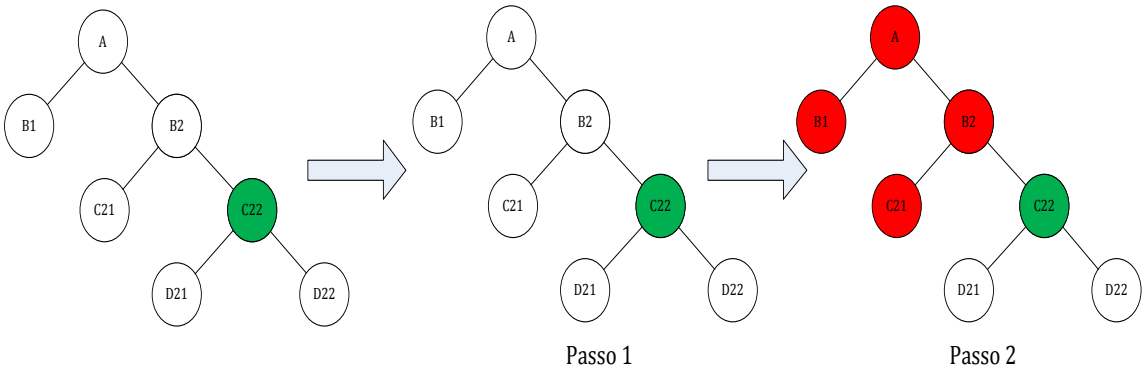


Figura 88 - Evolução do algoritmo para exemplo "Prático 2", com remoção de permissões

Neste caso, no passo 1 verifica-se através de uma pesquisa recursiva se existe alguma permissão originada nos papéis pai, mas como este papel representa a raiz da hierarquia, não possui papel pai fazendo o algoritmo convergir para o passo 2. Neste passo realiza-se uma pesquisa recursiva pelos papéis filho que não têm permissão, sendo apenas adicionados à lista de papéis a remover, os papéis A, B1, B2 e C21. A pesquisa recursiva pelo ramo na direita termina após a verificação que existe permissão para o papel C22. Deste modo são obtidas as informações sobre as políticas, necessárias ao gestor de políticas para informar o gestor de negócios.

3.3.4 Gestor de Políticas

O gestor de políticas realiza a gestão:

- Do *login* dos utilizadores no sistema;
- Das políticas de controlo de acesso;
- Das alterações efetuadas às políticas.

Todo o fluxo de informação entre o gestor de negócios e o servidor de políticas passa pelo gestor de políticas. Este decide quais as políticas a que os utilizadores do sistema têm acesso e reporta ao gestor de negócios as alterações que ocorrem nas

políticas de controlo de acesso. Pretendemos criar um servidor que permita responder a um número elevado de clientes. Deste modo foi construído um servidor mult-threaded, que cria uma thread (Worker) por cada pedido de comunicação, como ser visto pela Figura 89.

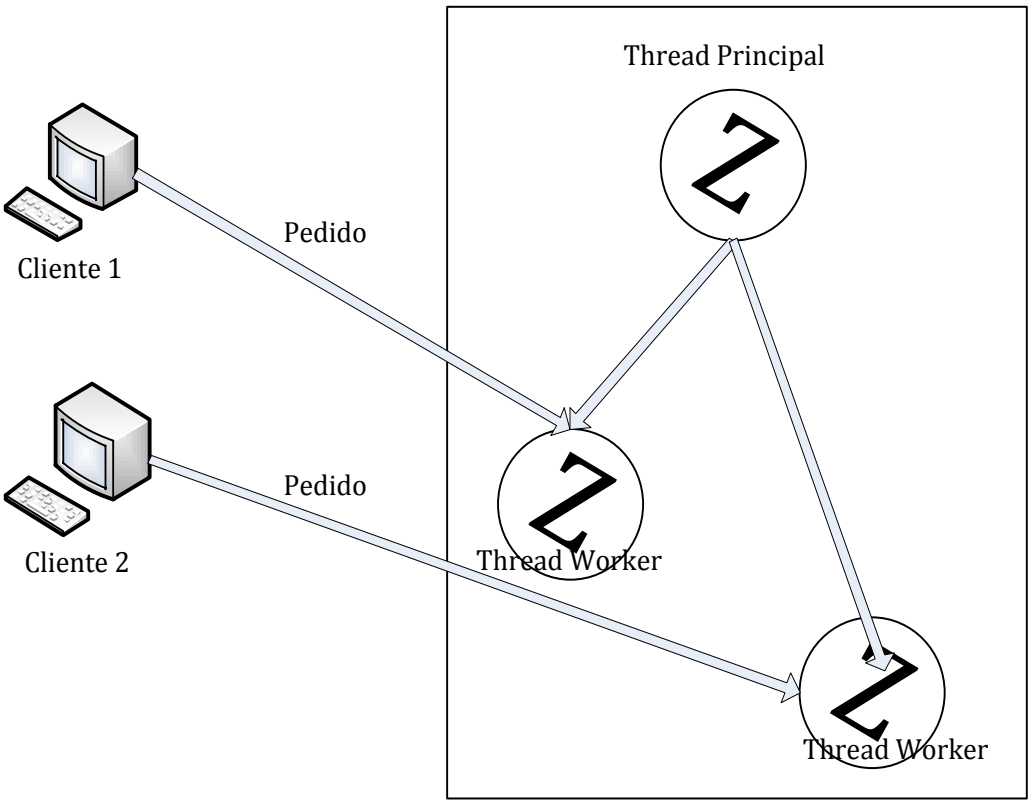


Figura 89 - Exemplo de comunicação entre clientes e servidor multi-thread

Um servidor multi-thread pode aceitar uma conexão de um cliente, começar uma thread para tratar aquela comunicação e continuar à escuta de novos pedidos de outros clientes [Oracle, '12n]. Esta vantagem deriva das threads serem uma sequência de instruções que correm independentemente de outro programa ou de outras threads.

As vantagens de um servidor multi-thread em relação a um servidor singlethread são as seguintes [Jenkov, '12]:

- Menos tempo é gasto fora da chamada ao método accept (método que permite aceitar uma conexão de um socket);
- Clientes que pretendam uma longa utilização do servidor não bloqueiam os outros clientes;

- Com a utilização de processadores multi-core, obtém-se uma maior performance.

Os pedidos realizados ao gestor de políticas são originados:

- No gestor de negócios por parte dos utilizadores que utilizam o ADCA, para efetuar o *login* no sistema e obter as informações necessárias para construir a lógica de negócio;
- No monitor de políticas, que é despoletado aquando de alterações nas políticas.

Para a criação desta arquitetura várias mensagens foram definidas para a comunicação. entre os vários elementos que a compõem. Estas mensagens são descritas no capítulo 3.2.

3.3.5 Monitor de Políticas

O monitor de políticas envia a informação sobre as alterações realizadas às políticas através de uma aplicação dll em C#. Uma Dll (biblioteca de vinculo dinâmico) [Microsoft, '12f; Microsoft, '12j] é uma biblioteca que contém código e dados, que podem ser executados por mais que um programa ao mesmo tempo. A utilização de dll's permite a um programa ser modularizado em componentes separados. O uso do dll traz outras vantagens tais como [Microsoft, '12j]:

- Utilização de menos recursos: como são feitas várias chamadas ao monitor de políticas e como este se encontra em memória, evita-se a duplicação do código;
- Facilita a alteração e instalação: como o dll vai ser instalado no servidor, neste caso servidor Sql Server, existe um único local onde será armazenado, permitindo assim a sua fácil atualização.

O DLL criado possui um método para envio de informação, a sua implementação em C# encontra-se na Figura 90.

```
public class PolicyWatcher
{
    public static void SendMessage(String ServerIp, int ServerPort, String Message)
    {
        if (ServerIp != null)
        {

            try
            {
                TcpClient tcpclnt = new TcpClient();

                tcpclnt.Connect(ServerIp, ServerPort);
                string str = Message;
                Stream stm = tcpclnt.GetStream();
                ASCIIEncoding asen = new ASCIIEncoding();
                byte[] ba = asen.GetBytes(str);

                stm.Write(ba, 0, ba.Length);
                tcpclnt.Close();
            }
            catch (Exception e)
            {
                throw new Exception("Error could connect to server!");
            }
        }
    }
}
```

Figura 90 - Implementação do Monitor de Políticas

Podemos verificar que no método SendMessage é realizada uma ligação ao gestor de políticas, através do parâmetro *ServerIp* (ip do gestor de políticas) e do parâmetro *ServerPort* (porta do gestor de políticas). A mensagem que possui a informação sobre a alteração ocorrida é também passada através do parâmetro *Message.* Depois da construção do dll que permite comunicar com o servidor de políticas, necessitamos de o instalar na base de dados. Para tal são realizados os seguintes passos:

- Criação de um objeto ASSEMBLY na base de dados através do comando `CREATE ASSEMBLY` [Microsoft, '12c]. Este comando permite adicionar esta aplicação como um objeto numa instância do sql server. Na Figura 91 podemos verificar o conjunto de comandos necessários para criação de um objeto assembly designado por *PolicyWatcher*.

```
CREATE ASSEMBLY PolicyWatcher
AUTHORIZATION dbo
FROM 'C:\DissJars\PolicyWatcher.dll'
```

Figura 91 - Comandos para criação de uma biblioteca dll no sql server

- Criação de uma *stored procedure* que permita a execução do dll. As *Stored-Procedures* são funcionam como procedimentos em linguagens de programação, pois [Microsoft, '12n]:
    - Aceitam vários parâmetros de entrada e retornam vários valores na forma de parâmetros de saída;
    - Permitem operações de base de dados, incluindo a chamada a outros procedimentos;
    - Retornam um valor de estado, indicando sucesso ou falha.

  A stored procedure é criada usando o comando `CREATE PROCEDURE`.

```
CREATE PROCEDURE SendMessage(@serverip nvarchar(50),@serverPort int,@message nvarchar(max))
AS EXTERNAL NAME PolicyWatcher.[PolicyWatcher.PolicyWatcher].SendMessage
GO
GO
```

Figura 92 - Comandos para criação da stored procedure que utiliza o método da biblioteca dll

Na Figura 92 pode-se verificar a criação de uma stored procedure relativa ao método *SendMessage,* que pertence ao objeto assembly *PolicyWatche*r. Seguindo estes passos, o dll é instalado no servidor de políticas. Necessitamos agora de realizar a monitorização das políticas e chamar a *stored-procedure* no caso de ocorrência de alterações. A solução passou pela utilização de triggers [Microsoft, '12p]. Estes são stored procedures especiais que podem ser executados automaticamente quando um utilizador tenta modificar dados específicos em tabelas específicas. Deste modo podemos saber em que tabela e que dados que foram alterados, realizando uma correta monitorização das permissões. Esta monitorização é realizada por quatro triggers:

- Dois triggers que verificam a inserção de autorizações ou de delegações;
- Dois triggers para verificar a remoção de autorizações ou de delegações.

Quando um destes triggers é despoletado, uma mensagem é enviada ao gestor de negócios a sinalizar a ocorrência de alterações nas políticas, apresentada no ponto 3.2.2.

4 Prova de Conceito

Neste capítulo apresenta-se uma aplicação com interface gráfico, que permite a execução de serviços de negócio e a visualização em tempo de execução dos esquemas que compõem a lógica de negócio através de um diagrama incorporado na própria interface. De modo a facilitar a correta configuração do servidor de políticas foram também desenvolvidas aplicações de apoio. Nos próximos parágrafos descreve-se a forma como o servidor de políticas é configurado através das aplicações de apoio e o funcionamento da aplicação principal onde os serviços de negócio são executados.

4.1 Esquema Geral e Preparação do Ambiente

Para o desenvolvimento desta aplicação foi utilizada a base de dados pública NorthWind [Microsoft, '12i], que apresenta tabelas referentes a venda de produtos, tais como, clientes, empregados, categoria de produtos e fornecedores.

Nesta aplicação foram utilizadas as seguintes tabelas:

- Categories: que corresponde às diferentes categorias dos produtos;
- Products: que corresponde às informações relativas aos variados produtos;
- Suppliers: que corresponde às informações relativas aos fornecedores.

Utilizando esta base de dados pública, criámos um conjunto de papéis que contêm um conjunto de esquemas de negócio, permitindo a realização de diferentes ações por parte dos utilizadores do sistema. Para esta aplicação foi desenvolvido o seguinte esquema de papéis:

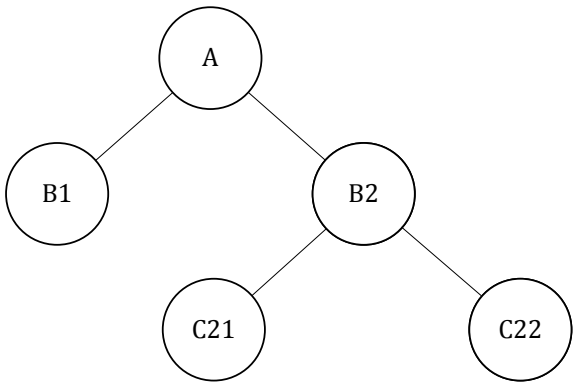


Figura 93 - Esquema de papéis geral utilizado na aplicação de teste

Com a construção desta estrutura hierárquica de três níveis, podemos verificar a propagação das várias permissões atribuídas aos utilizadores no nosso diagrama.

Os esquemas de negócio desenvolvidos são os seguintes:

- IPrd_s- Este esquema de negócios utiliza a tabela Products, e permite a leitura de todos os campos desta tabela para acesso a informação sobre produtos (Ver Figura 94). Permite também a seleção por id do produto ou por id do fornecedor (supplier);

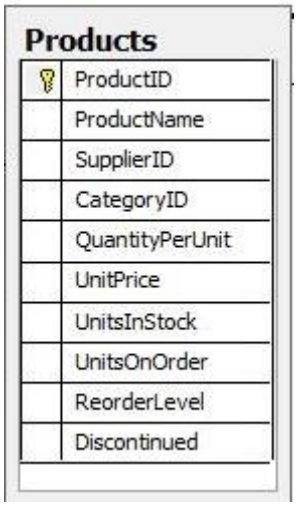


Figura 94 - Tabela Products com os respetivos campos

- ISup_s- Este esquema de negócio utiliza a tabela suppliers e permite a leitura de todos os campos desta tabela para acesso a informação sobre os fornecedores (Ver Figura 95);

Figura 95 - Tabela Suppliers com os respetivos campos

- ICat_s- Este esquema de negócio utiliza a tabela categories e permite a leitura de todos os campos desta tabela (Ver Figura 96). Disponibiliza métodos para a inserção, atualização de campos e remoção de linhas da tabela. Podemos selecionar para visualização todas as entradas da tabela ou apenas as entradas com id's específicos;

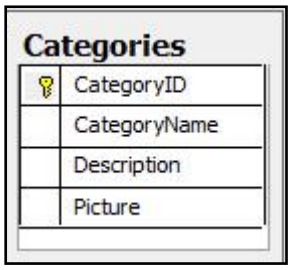


Figura 96 - Tabela Categories com os respetivos campos

- ICat_i- Este esquema de negócio é utilizado para introdução de uma nova categoria através da utilização de uma expressão CRUD do tipo Insert.

Ao conjunto de papéis definidos anteriormente (Ver Figura 93) são atribuídos os esquemas de negócio especificados, de acordo com a Tabela 19.

Papel	**BS**	**Crud**	
		Ref	**Expressão**
Role_B1	ICat_i	all	Insert into Categories Values(?,?,?)
Role_B2	IPrd_s	all	Select * From Products
		byId	Select * From Products where productId=?
		bySupplierId	Select * From Products where supplierId=?
Role_C21	ICat_s	all	Select * From Categories
		byId	Select * From Categories where categoryId=?
Role_C22	ISup_s	all	Select * From Suppliers

Tabela 19 - Tabela com os papéis e os respetivos esquemas de negócio e expressões CRUD utilizadas no desenvolvimento da aplicação

Legenda:

BS - Esquema de Negócio; Ref- Referência da expressão CRUD; Expressão- Expressão CRUD

Foram criados três utilizadores que podem utilizar os diferentes papéis de acordo com as permissões atribuídas. Para utilização dos serviços de negócio procedeu-se à configuração do ambiente (esquemas de negócios, papéis, expressões CRUD) no servidor de políticas. Para esta finalidade criou-se um conjunto de aplicações de apoio à configuração, as quais são descritas nos próximos parágrafos.

4.1.1 Criação do Servidor de Políticas

Para criação do servidor de políticas foi criado um script SQL que permite inserir as tabelas designadas no ponto 3.3.3.2.2. Este script, para além da criação destas tabelas, cria também os triggers que permitem chamar o monitor de políticas para envio de informações sobre as alterações de políticas.

Configuração do Servidor de Políticas

A configuração do servidor de políticas é realizada através de uma aplicação em java, designada por Configurador de Segurança. Esta aplicação facilita a introdução de informações no servidor de políticas através da criação de um conjunto de interfaces. Estas interfaces contêm informação, sobre aplicações, papéis, esquemas de negócio e expressões CRUD. Com estas interfaces, o configurador de segurança cria entradas nas tabelas correspondentes, conseguindo de uma maneira fácil e eficiente, introduzir as informações requeridas para a utilização da aplicação de teste. A configuração do servidor de políticas é realizada através dos seguintes passos:

- Inicialmente é construída uma interface que contém as expressões Crud que queremos utilizar. Para o nosso caso e de acordo com as especificações do ponto 4.1 foi criada a interface que se encontra na Figura 97.

```
public interface ICrud {
    String iCat_all="Insert into Categories values (?,?,?);";
    String sPrd_all="Select * from Products";
    String sPrd_byId="Select * from Products where ProductId=?";
    String sPrd_bySupplierId="Select * from Products where supplierId=?";
    String sPrdCat_byCategoryId="Select p.*,c.categoryName,c.Description " +
                                "from Products p, Categories c " +
                                "where p.CategoryID=c.CategoryID";
    String sCat_all="Select * from Categories";
    String sCat_byId="Select * from Categories where categoryId=?";
    String sSup_all="Select * from Suppliers";
}
```

Figura 97 - Interface com a declaração dos CRUDs

- O próximo passo é a construção dos interfaces que armazenam a informação sobre os papéis. Estes interfaces devem indicar qual o próximo papel na hierarquia. Para tal, utilizamos uma composição hierárquica das interfaces através do comando *extends* do java. Um exemplo desta interface encontra-se na Figura 98.

```
public interface IRole_A extends IRole_B2, IRole_B1{
}
```

Figura 98 - Declaração da interface para o papel A

Podemos verificar que a interface do papel A possui referências para os respetivos papéis filhos (B1 e B2) através da utilização do comando *extends*. As informações associadas ao papéis tais como esquemas de negócio e expressões Crud são também inseridas nestas interfaces. Para tal é criado inicialmente um objeto do tipo Class que referência o esquema de negócio e de seguida são colocadas as expressões Crud respetivas através de um array de objetos do tipo String. Este caso pode ser verificado na Figura 99, na qual se representa a interface referente ao papel B1. Este papel, como especificado em 4.1, recebe o esquema de negócio *ICat_i* que por sua vez possui acesso à expressão Crud, cuja referência é *iCat_all*;

```
public interface IRole_B1 {

  public static final Class<ICat_i> icat_i=ICat_i.class;
  public static final String[] crud_icat_i = new String[] { ICrud.iCat_all };

}
```

Figura 99 - Exemplo de interface para o papel B1

- Em seguida necessitamos de uma interface principal para representar a aplicação e o conjunto de interface que a compõem. Isto é realizado com a construção de uma interface que possui a referência para a aplicação e o conjunto de papéis raiz que compõem a hierarquias de papéis. Um exemplo desta interface encontra-se na Figura 100.

```
public interface IApplication {
    String app = "app";
    Class[] roles = new Class[] {
        IRole_A.class};
}
```

Figura 100 - Interface principal da aplicação

Podemos verificar que é criada uma referência para a aplicação designada por "app" e também é armazenada a interface referente ao papel raiz A. O configurador de segurança consegue obter todos os outros papéis através do papel raiz com a utilização da API Reflection, sendo por isso criada apenas a referência para o papel raiz. Deste modo constroem-se as interfaces necessárias ao Configurador de Segurança para a configuração do servidor de políticas. Podemos então chamar o Configurador de Segurança. Este contém dois métodos para configuração:

- O método ConfigureServer: onde definimos o nome de utilizador, a password e o url para ligação ao servidor de políticas;
- O método Configure: permite a configuração do servidor de políticas através das interfaces desenvolvidas.

```
public class SecurityConfiguratorMain {

    public static void main(String[] args) throws SCException {

        String url = "sqlserver://127.0.0.1:1433;database=PolicyServer;";
        SecurityConfigurator sc=new SecurityConfigurator();

        sc.ConfigureServer("sa", "sqlnovo", url);

        sc.Configure(IApplication.class,ICrud.class);
    }
}
```

Figura 101 - Configuração do Configurador de Segurança

Na Figura 101 encontrámos um exemplo de utilização destes dois métodos. Podemos verificar que no método Configure são passadas a interface da aplicação e a interface onde as expressões CRUD são especificadas. Com a utilização desta aplicação de

apoio conseguimos inserir a informação necessária no servidor de políticas de uma forma fácil e eficiente.

4.1.2 Extrator de Políticas

O extrator de políticas é a aplicação de apoio que permite obter um ficheiro jar, com toda a informação sobre uma aplicação para permitir o desenvolvimento facilitado de uma aplicação que utilize a ADCA.

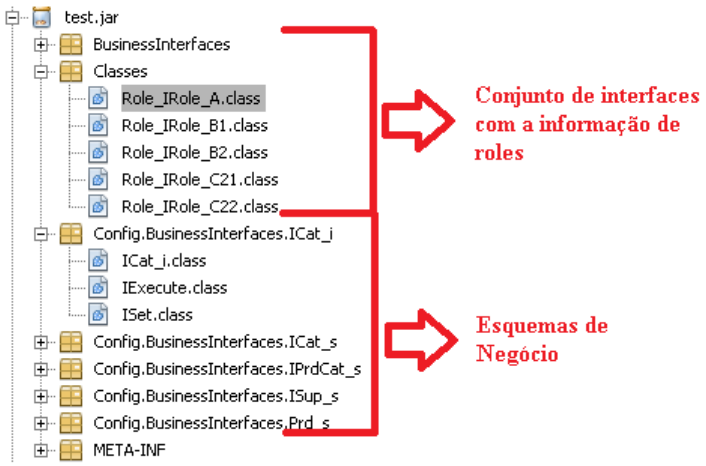


Figura 102 - Esquema interno do ficheiro jar resultante do extrator de políticas

Na Figura 102 apresenta-se o esquema interno do ficheiro jar gerado. Podemos verificar a existência dos vários esquemas de negócio desenvolvidos e o conjunto de interfaces contendo a informação sobre os vários papéis, nomeadamente, os esquemas de negócio utilizados e um inteiro representativo do id da expressão CRUD. Na Figura 103 podemos encontrar um exemplo de uma interface obtida. Neste caso o papel representado é o papel B1 cuja especificação se encontra no ponto 4.1. Nesta interface existem objetos Class que representam os esquemas de negócio e variáveis numéricas nas quais são armazenados os ids(tal como no servidor de políticas) das expressões CRUD associadas. Com este arquivo jar o desenvolvimento torna-se mais facilitado pois:

- Podemos verificar para cada papel qual o tipo de esquemas de negócio associados;
- Escolher a expressão CRUD correta através do id que é especificado nas interfaces dos papéis.

```
package Classes;

import Config.BusinessInterfaces.IPrdCat_s.IPrdCat_s;
import Config.BusinessInterfaces.Prd_s.IPrd_s;

public interface Role_IRole_B2 extends Role_IRole_C21, Role_IRole_C22 {

    public static final Class<IPrdCat_s> prdcat_s;
    public static final int prdcat_s_sPrdCat_byCategoryId = 1;
    public static final Class<IPrd_s> prd_s;
    public static final int prd_s_sPrd_all = 2;
    public static final int prd_s_sPrd_byId = 3;
    public static final int prd_s_sPrd_bySupplierId = 4;
}
```

Figura 103 - Exemplo de interface utilizada na especificação dos papéis

A interface gráfica do extrator de políticas encontra-se na Figura 104, nesta podemos configurar a conexão para ligação à base de dados no painel Conexão. Nos campos à esquerda define-se a referência da aplicação que queremos obter e a localização onde o ficheiro jar será criado.

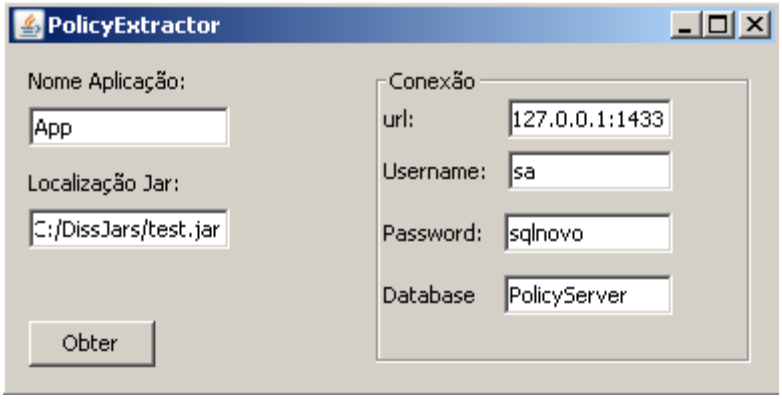


Figura 104 - Interface de interação com o utilizador do extrator de políticas

4.1.3 Gestor de Segurança

O Gestor de Segurança é uma aplicação de apoio que permite a alteração das políticas em tempo de execução. Permite atribuir ou remover autorizações e delegações referentes ao conjunto de papéis especificados.

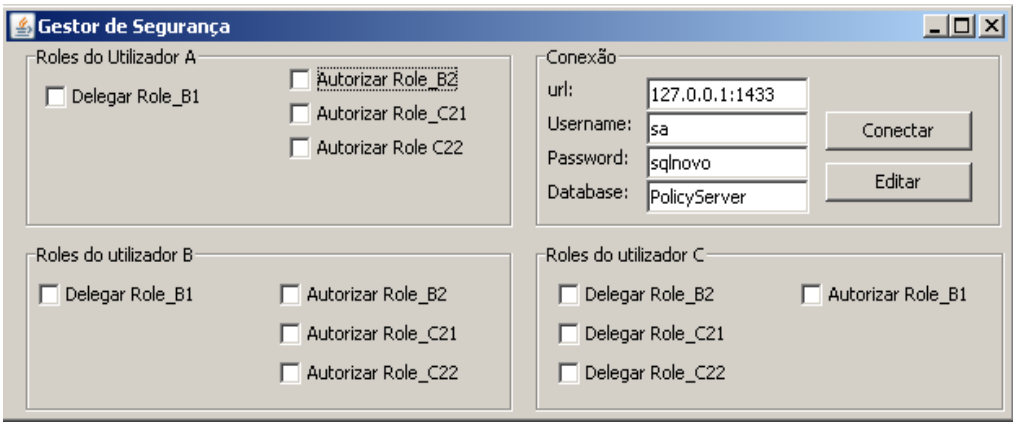


Figura 105 - Interface gráfica da aplicação Gestor de Segurança

Na Figura 105 encontramos a interface gráfica do gestor de segurança. Nesta podemos verificar a existência de quatro painéis. No painel Conexão é configurada a conexão à base de dados, enquanto nos restantes painéis utilizamos objetos do tipo *checkbox* que permitem atribuir ou retirar autorizações ou delegações referentes aos papéis dos três utilizadores.

4.2 Utilização da Aplicação de Teste

Neste capítulo apresenta-se a aplicação desenvolvida que permite realizar o teste da ADCA. Esta aplicação foi desenvolvida em java e c# e permite a interação com as políticas de acesso armazenadas no servidor de políticas. A interface principal da nossa aplicação é apresentada na Figura 106.

Nesta interface encontramos as seis secções principais:

- Na secção 1 são escolhidos os esquemas de negócio e expressões CRUD a executar.
- Na secção 2 efetua-se a configuração dos campos para ligação à base de dados relacional.
- Na secção 3 apresenta-se um diagrama com o estado de permissão de acesso a papéis (a vermelho, a sua negação, a verde a sua autorização).
- Na secção 4 são atribuídos em tempo de execução, diferentes valores aos parâmetros dinâmicos que podem compor a expressão CRUD selecionada.
- Na secção 5 são apresentados os resultados da tabela resultante da execução da expressão CRUD (apenas se esta for do tipo Select).
- Na secção 6 apresentam-se os comandos adicionais no caso da interface selecionada ser a interface ICat_s, visto que esta interface permite a atualização, inserção e remoção de linhas resultantes da execução de um CRUD do tipo Select.

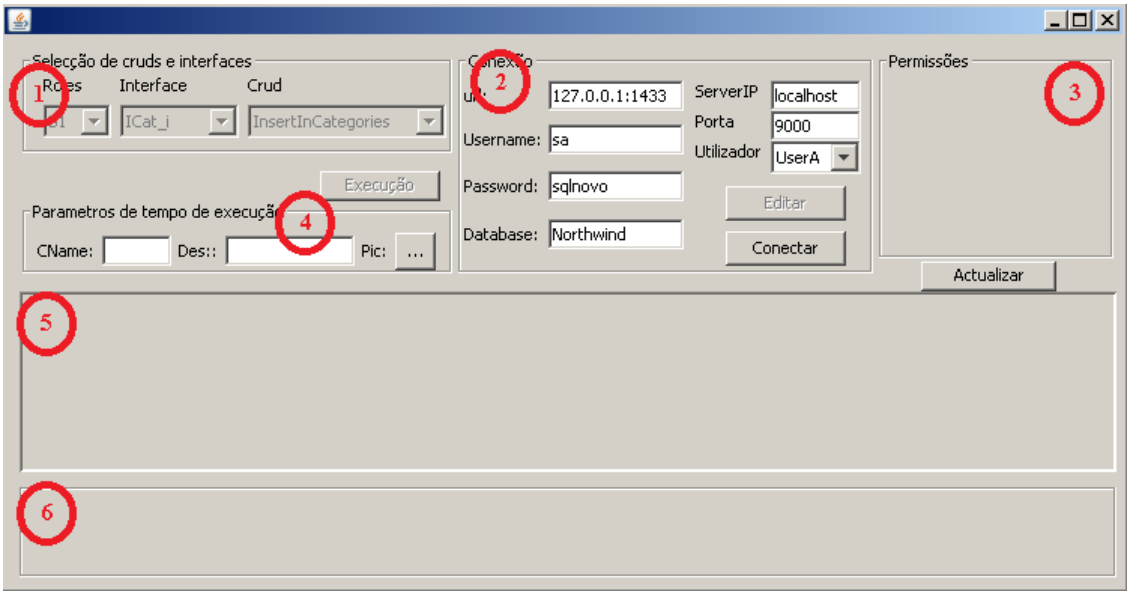


Figura 106 - Interface gráfica, de utilização da aplicação de teste

O esquema de utilização desta aplicação é o seguinte, representado, pela Figura 107.

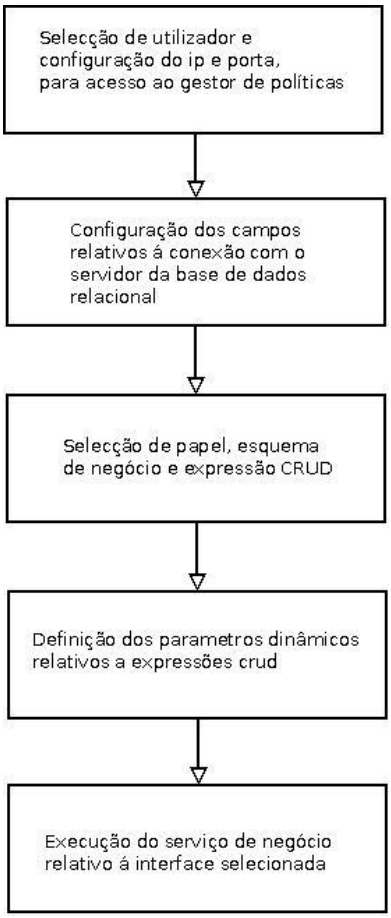


Figura 107 - Esquema principal de execução

Considere-se o caso em que o utilizador tem acesso a todos os papéis exceto o papel A1. Ao clicar no botão Atualizar da secção 3 é apresentado o seguinte diagrama da Figura 108.

No caso da execução do serviço de negócio correspondente ao esquema de negócio IPrd_s e da expressão CRUD relativa a todos os elementos, a secção 5 é preenchida com o resultado da execução da expressão na base de dados (Figura 109).

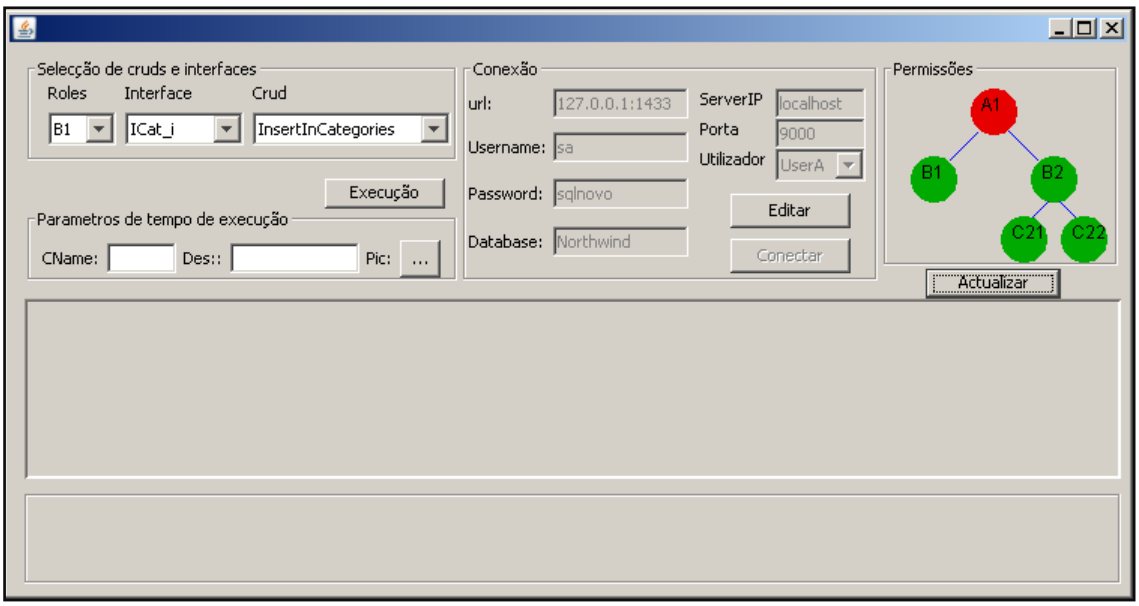


Figura 108 - Interface gráfica com a visualização do diagrama de permissões

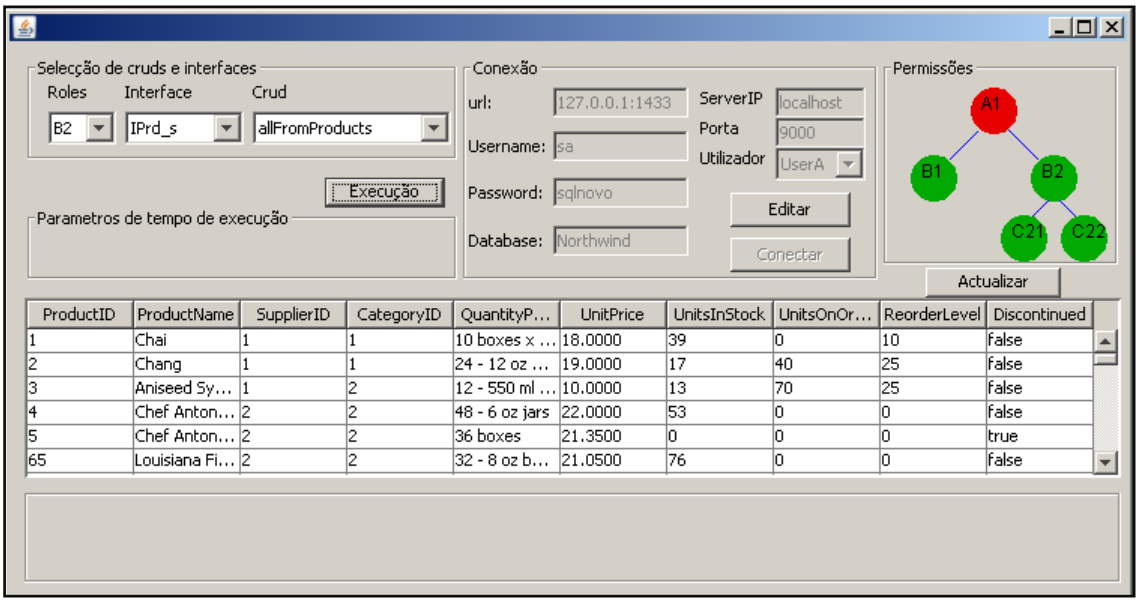


Figura 109 - Interface gráfica com a visualização dos resultados

A utilização de uma expressão CRUD à qual é necessária a atribuição de valores origina a criação de campos de texto em tempo de execução, onde o utilizador pode inserir os valores (Figura 110).

No caso de utilização do esquema de negócio ICat_s, podemos verificar a existência de comandos que possibilitam a alteração das várias linhas que representam a tabela como se pode ver no fundo da Figura 111.

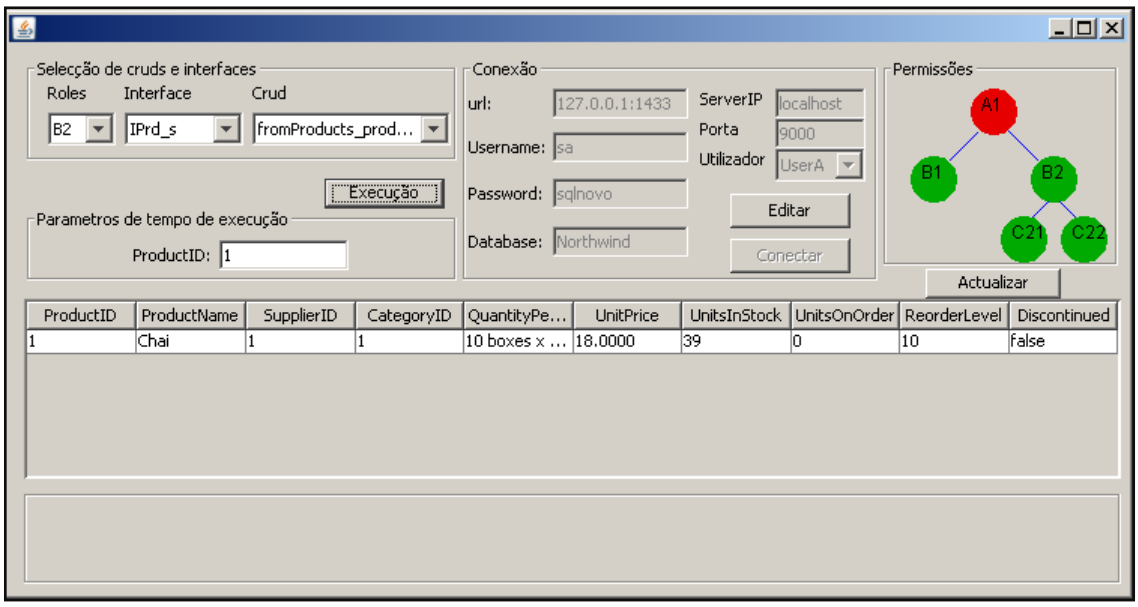


Figura 110 - Interface gráfica com campos para inserção de parâmetros do CRUD

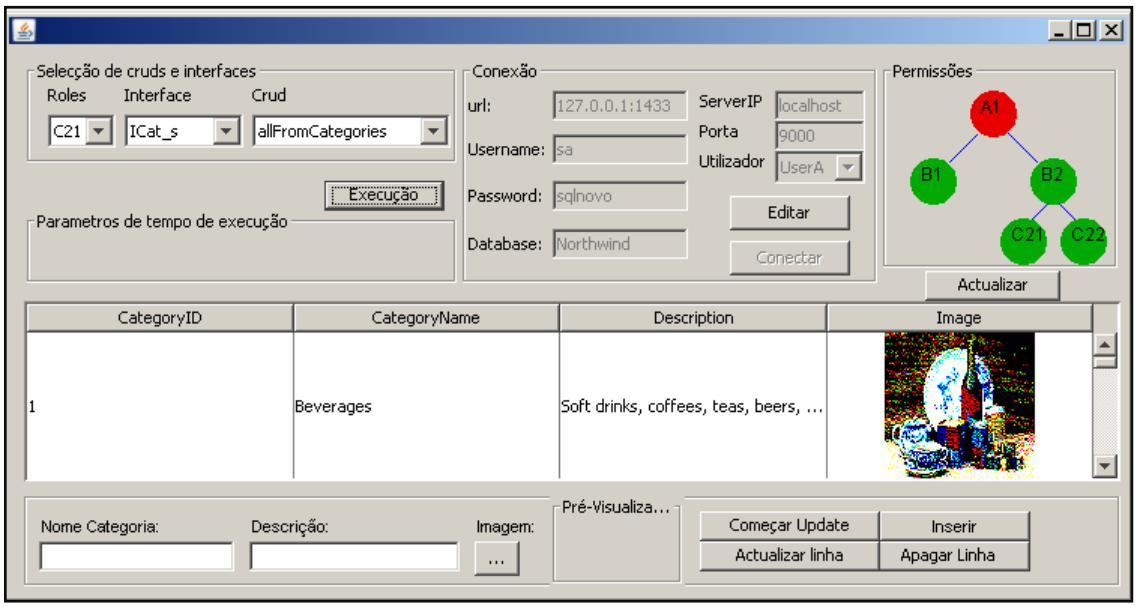


Figura 111 - Interface gráfica apresentado os botões adicionais para o esquema de negócio ICat_s

A execução de um esquema de negócio ao qual não se tem direito de acesso despoleta a apresentação de uma mensagem de autorização negada, como ser visto pela Figura 112. Podemos verificar pelo diagrama da figura que não existe permissão para acesso ao papel B2 e como tal está negado o acesso aos esquemas de negócio IPrd_s e ISup_s.

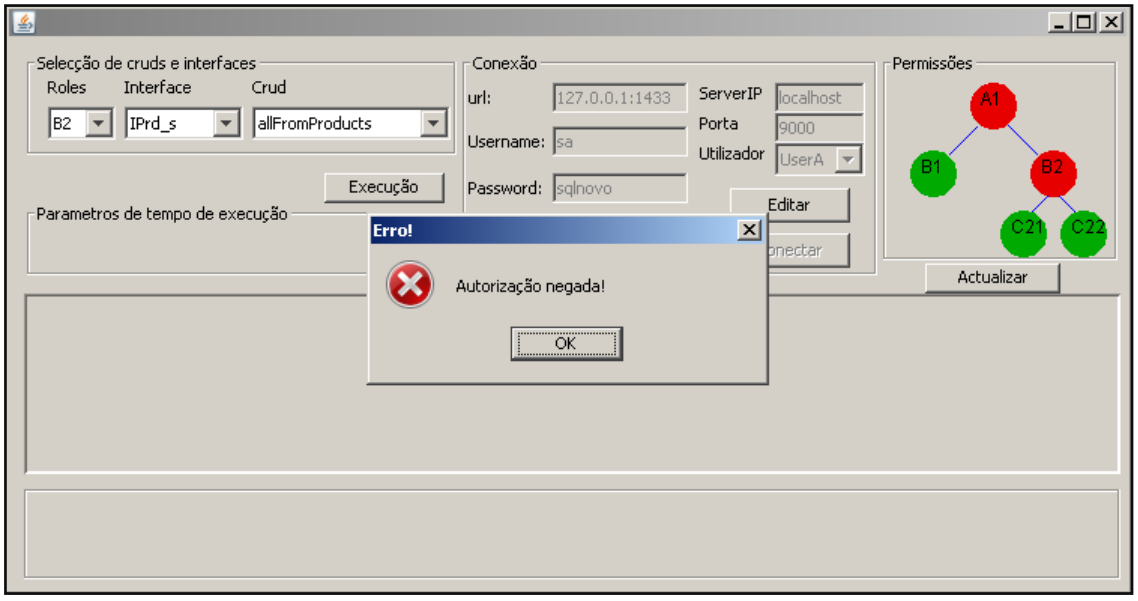


Figura 112 - Interface gráfica apresentado a mensagem de Autorização Negada

Criamos assim um conjunto de aplicações que apoiam o administrador na implementação do servidor de políticas e que permitem verificar o correto funcionamento da ADCA.

5 Conclusão

O trabalho apresentado nesta dissertação abordou o aspeto da construção de uma lógica de negócio que utiliza o software existente para interação com base de dados (JDBC e ADO.net) ao mesmo tempo que é aplicado controlo de acesso a informação sensível. Utilizando as bibliotecas disponibilizadas pela linguagem de programação Java, como Reflection e bibliotecas para carregamento de classes em tempo de execução, foi possível atingir os objetivos propostos. A utilização de ficheiros do tipo jar para armazenamento das interfaces referentes aos esquemas de negócio foi essencial para uma gestão mais eficaz da lógica de negócio. Embora as aplicações desenvolvidas em java permitam a sua execução num amplo conjunto de sistemas operativos, desenvolveu-se uma versão estática em C# da parte cliente da arquitetura. Isto permitiu demonstrar que os métodos propostos podem ser implementados utilizando várias tecnologias, neste caso o ADO.net.

Através da prova de conceito apresentada foi possível verificar o correto funcionamento da arquitetura desenvolvida. Embora existam muitas vantagens na utilização desta arquitetura esta também apresenta as suas desvantagens. Nos próximos parágrafos efetuamos uma discussão destas.

5.1 Escalabilidade, Disponibilidade e Adaptabilidade

A ADCA foi desenvolvida com o intuito de possibilitar a utilização a um grande número de utilizadores. Isto é possível com a utilização de uma entidade central, designada por gestor de políticas que recebe os pedidos dos clientes. Esta entidade central foi construída com recurso a threads de modo a responder a vários pedidos ao mesmo tempo. Podemos ter assim um grande número de utilizadores ligados ao sistema e cujos pedidos são tratados de imediato. Este sistema não foi concebido para ser utilizado por outras tecnologias, como por exemplo os *Web Services*. Visto que o gestor de negócios apenas implementa métodos que utilizam *software* intermédio para interação com base de dados, este teria que ser reescrito para possibilitar tais operações.

5.2 Problemas Comuns

Embora a ADCA ofereça diversas vantagens em relação às soluções correntes de sofware para interação com base de dados relacionais esta tem os seguintes problemas:

- É necessária a utilização de uma versão superior à Versão 1.6 do JAVA. Na versão JAVA 1.6 não existe forma de fechar a ligação ao arquivo jar, fazendo com que

depois do carregamento dos serviços de negócio este arquivo fique bloqueado não sendo possível a adaptação às alterações das políticas. Este problema é resolvido na versão 1.7 onde é disponibilizado um método que permite o fecho da ligação, possibilitando a adaptação dinâmica;

- Depois do carregamento de um serviço de negócio, este contínua em memória na JVM mesmo depois da adaptação dinâmica da lógica de negócio. Isto poderia levar à execução de serviços para os quais não existe autorização. Este problema é resolvido na nossa aplicação de teste através de um método que verifica a existência do serviço na lógica de negócio criada. No caso da não existência do serviço é retornada uma exceção impedindo a sua utilização;
- O tempo que demora a construção da lógica de negócio para os utilizadores aumenta consoante o número de esquemas de negócio e a complexidade inerente destes. Uma forma de resolver este problema seria a modularização de aplicações possibilitando assim, a diminuição do conjunto de esquemas de negócio que seriam criados pelo gestor de negócios.

5.3 Trabalho Futuro

A ADCA na atual versão, apresenta uma primeira abordagem em resposta aos problemas apresentados no ponto 1. O modelo construído para armazenamento das políticas de acesso é baseado no modelo rbac1, o qual permite a utilização de uma hierarquia de papéis e de delegações. Num desenvolvimento futuro poderia ser utilizado o modelo rbac3 com possibilidade de utilização de restrições. As restrições permitem por exemplo, impedir a utilização de um papel em certos períodos temporais, importante em certas situações.

A criação de esquemas de negócio a nível programático pode ser demorada e requer alguma experiência em linguagens de programação por parte do administrador. Esta dificuldade pode ser suavizada pela introdução de uma aplicação com interface gráfico que permite ao administrador definir qual a expressão CRUD que quer utilizar e quais os campos que pretende ler, actualizar, inserir ou apagar em caso de expressão do tipo Select. A interface gráfica que se propõe é baseada no gestor de expressões CRUD dos papers [Pereira, '10; Pereira, '11b]. Enquanto esta é baseada apenas na construção de esquemas de negócio com todos os campos, propomos a criação de um gestor de CRUDS que permita a criação de esquemas de negócio com controlo de acesso, onde o administrador específica quais os campos que pretende a que se tenha acesso. Esta interface gráfica poderia ter as seguintes secções:

- Secção para inserção de expressão CRUD e preparação dos parâmetros da mesma para realizar a sua execução na base de dados relacional. Na Figura 113 temos um exemplo de como esta secção seria;

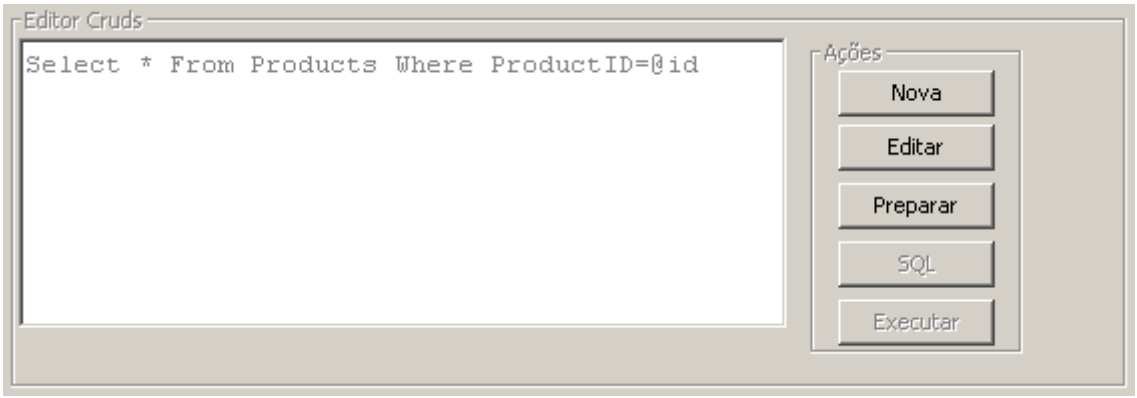


Figura 113 - Secção exemplo para introdução de expressões CRUD

- Secção para seleção de atributos e do tipo de acesso atribuído. Um exemplo desta secção encontra-se na Figura 114. Neste caso utilizamos um conjunto de caixas de verificação em que se atribuem e retiram permissões de acesso a campos em tabelas.

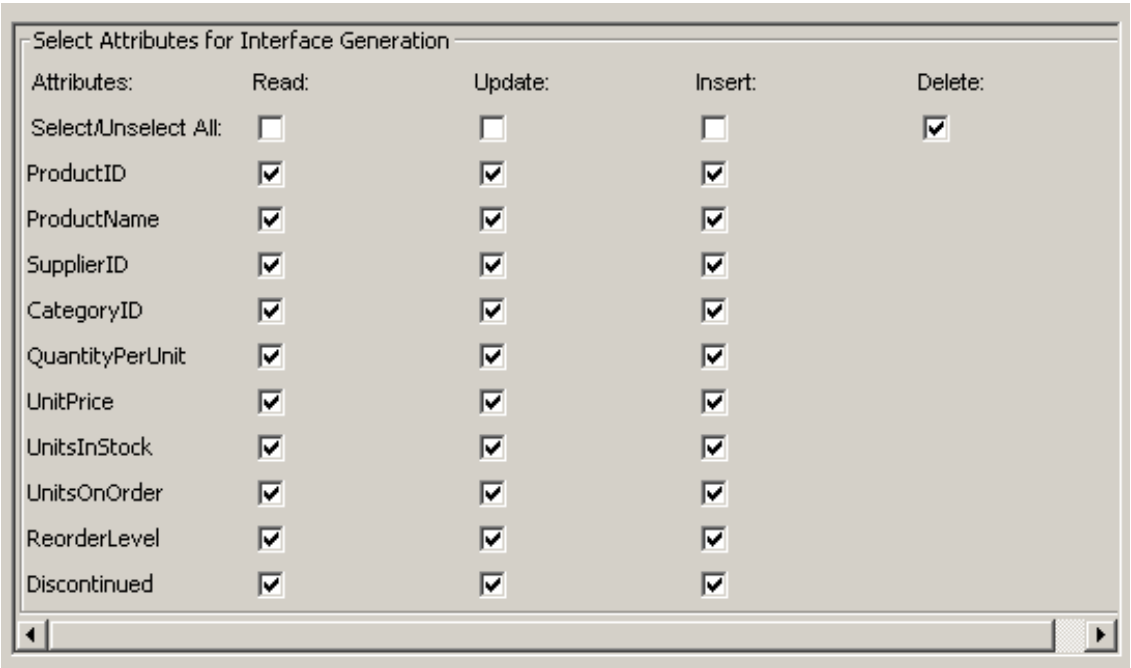


Figura 114 - Secção exemplo para seleção de permissões em expressões do tipo Select

Com esta aplicação o administrador teria um trabalho mais facilitado e menos demorado na construção dos esquemas de negócio.

A nível de segurança este trabalho pode ser melhorado de modo a existir encriptação de mensagens que são transmitidas entre os vários componentes que compõem a nossa arquitetura e a criação de proteção do sistema contra situações de erro. Por exemplo a realização da sincronização de clientes com as políticas definidas, no caso de o cliente se desconectar momentaneamente do sistema e durante esse período serem realizadas alterações nas políticas. A resolução deste caso podia passar, pela criação de uma tabela adicional no sistema, na qual se expressa o estado em que o cliente se encontra, e a criação de uma thread no gestor de negócios que comunica periodicamente com o gestor de políticas para este verificar o seu correto estado. No caso de estado incorreto por parte de um cliente, o gestor de políticas enviava todas as alterações que foram efetuadas durante o tempo em baixa do cliente.

6 Bibliografia e Referências

[Barka, '04] Barka, Ezedin and Sandhu, Ravi: "Role-Based Delegation Model/Hierarchical Roles (RBDM1)." Proceedings of the 20th Annual Computer Security Applications Conference, IEEE Computer Society(2004), 396-404.

[Barka, '00] Barka, Ezedin, Sandhu, Ravi and S, Ravi: "A Role-Based Delegation Model and Some Extensions." Information Systems Security, (2000).

[Bell, '73] Bell, D. and La Padula, L.: "Secure computer systems: A mathematical model"; MITRE Corp, (1973).

[Biba, '77] Biba, K.J. (1977). Integrity considerations for secure computer systems. M. Corp.**:** 76-372.

[Blosser, '00] Blosser, Jeremy. (2000). "Explore the Dynamic Proxy API." from http://www.javaworld.com/jw-11-2000/jw-1110-proxy.html.

[Brant, '00] Brant, John and Yoder, Joseph: "Creating Reports with Query Objects." Pattern Languages of Programs, University of Illinois at Urbana, Champaign (2000).

[Capitani di Vimercati, '07] Capitani di Vimercati, Sabrina, Foresti, Sara and Samarati, Pierangela (2007). Authorization and Access Control.

Security, Privacy, and Trust in Modern Data Management. M. Petković and W. Jonker, Springer Berlin Heidelberg**:** 39-53.

[Codd, '83] Codd, E. F.: "A relational model of data for large shared data banks." Commun. ACM, 26, 1 (1983), 64-69.

[Corcoran, '09] Corcoran, Brian J., Swamy, Nikhil and Hicks, Michael: "Cross-tier, label-based security enforcement for web applications." Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, ACM, Providence, Rhode Island, USA (2009), 269-282.

[Damiani, '05] Damiani, Ernesto, Vimercati, Sabrina De Capitani di and Samarati, Pierangela: "New Paradigms for Access Control in Open Environments." Dipartimento di Tecnologie dell'Informazione(2005).

[Green, '12] Green, Date. (2012). "The Reflection API what can we do." from http://rangiroa.essi.fr/cours/turorial%20java/reflect/index.html.

[Hansson, '04] Hansson and Heinemeier, David. (2004). "Ruby on Rails." from http://rubyonrails.org/.

[Java2s, '12] Java2s. (2012). "Transaction Isoltation Levels.", from http://www.java2s.com/Tutorial/Java/0340__Database/JDBCTransactionIsolationLevels.htm.

[JBOSS, '12] JBOSS. (2012). "HIBERNATE." from http://docs.jboss.org/hibernate/orm/4.1/quickstart/en-US/html/.

[Jenkov, '12] Jenkov, Jakob. (2012). "Multithreaded Server in Java." from http://tutorials.jenkov.com/java-multithreaded-servers/multithreaded-server.html.

[Lampson, '74] Lampson, Butler W.: "Protection." SIGOPS Oper. Syst. Rev., 8, 1 (1974), 18-24.

[Meijer, '06] Meijer, Erik, Beckman, Brian and Bierman, Gavin: "LINQ: reconciling object, relations and XML in the .NET framework." Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM, Chicago, IL, USA (2006), 706-706.

[Microsoft, '12a] Microsoft. (2012a). "C Sharp." from http://msdn.microsoft.com/library/z1zx9t92.

[Microsoft, '12b] Microsoft. (2012b). "C# and Java: Comparing Programming Languages." from http://msdn.microsoft.com/en-us/library/ms836794.aspx.

[Microsoft, '12c] Microsoft. (2012c). "CREATE ASSEMBLY." from http://msdn.microsoft.com/en-us/library/ms189524.aspx.

[Microsoft, '12d] Microsoft. (2012d). "DataSet." from http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx.

[Microsoft, '12e] Microsoft. (2012e). "HashTable." from http://msdn.microsoft.com/en-us/library/system.collections.hashtable.aspx.

[Microsoft, '12f] Microsoft. (2012f). "How to: Create and Use C# DLLs." from http://msdn.microsoft.com/en-us/library/3707x96z(v=vs.80).aspx.

[Microsoft, '12g] Microsoft. (2012g). "Linq." from http://msdn.microsoft.com/en-us/library/bb397926.aspx.

[Microsoft, '12h] Microsoft. (2012h). ".NET Platform." from http://msdn.microsoft.com/en-us/library/zw4w595w.aspx.

[Microsoft, '12i] Microsoft. (2012i). "NorthWind." from http://www.microsoft.com/en-us/download/details.aspx?id=23654.

[Microsoft, '12j] Microsoft. (2012j). "O que é uma DLL?", from http://support.microsoft.com/kb/815065/pt-br.

[Microsoft, '12k] Microsoft. (2012k). "ODBC." from http://support.microsoft.com/kb/110093.

[Microsoft, '12l] Microsoft. (2012l). "Overview of ADO.net." from http://msdn.microsoft.com/en-us/library/h43ks021(v=vs.71).aspx.

[Microsoft, '12m] Microsoft. (2012m). "Sql Server." from http://msdn.microsoft.com/en-us/library/ms166352(v=sql.90).aspx.

[Microsoft, '12n] Microsoft. (2012n). "Stored Procedure." from http://msdn.microsoft.com/en-us/library/ms187926.aspx.

[Microsoft, '12o] Microsoft. (2012o). "Transactions." from http://msdn.microsoft.com/en-us/library/ms190612(v=sql.105).aspx.

[Microsoft, '12p] Microsoft. (2012p). "Triggers." from http://msdn.microsoft.com/en-us/library/aa258254(v=sql.80).aspx.

[Microsoft, '12q] Microsoft. (2012q). "Visual Studio." from http://www.microsoft.com/visualstudio/en-us.

[Morin, '10] Morin, Brice, Mouelhi, Tejeddine, Fleurey, Franck, Traon, Yves Le, Barais, Olivier and J, Jean-Marc: "Security-driven model-based dynamic adaptation." Proceedings of the IEEE/ACM international conference on Automated software engineering, ACM, Antwerp, Belgium (2010), 205-214.

[Napes, '12] Napes. (2012). "Java + information flow." from http://www.napes.co.uk/blog/information-flow-java-jif/.

[Netbeans, '12] Netbeans. (2012). "Netbeans." from http://netbeans.org/.

[Oracle, '12a] Oracle. (2012a). "ClassLoader." from http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/ClassLoader.html.

[Oracle, '12b] Oracle. (2012b). "HashMap." from http://docs.oracle.com/javase/1.4.2/docs/api/java/util/HashMap.html.

[Oracle, '12c] Oracle. (2012c). "JarEntry." from http://docs.oracle.com/javase/6/docs/api/java/util/jar/JarEntry.html.

[Oracle, '12d] Oracle: "JarInputStream." (2012d).

[Oracle, '12e] Oracle. (2012e). "JarOutputStream." from http://docs.oracle.com/javase/1.4.2/docs/api/java/util/jar/JarOutputStream.html.

[Oracle, '12f] Oracle. (2012f). "Java." from http://en.wikipedia.org/wiki/Java_(programming_language).

[Oracle, '12g] Oracle. (2012g). "Java Archive." from http://docs.oracle.com/javase/tutorial/deployment/jar/.

[Oracle, '12h] Oracle. (2012h). "JavaCompiler." from http://docs.oracle.com/javase/6/docs/api/javax/tools/JavaCompiler.html.

[Oracle, '12i] Oracle. (2012i). "JDBC." from http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html.

[Oracle, '12j] Oracle. (2012j). "JPA." from http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html.

[Oracle, '12k] Oracle. (2012k). "Mapping Sql Types to Java." from http://docs.oracle.com/javase/1.5.0/docs/guide/jdbc/getstart/mapping.html.

[Oracle, '12l] Oracle. (2012l). "PreparedStatement." from http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html.

[Oracle, '12m] Oracle. (2012m). "ResultSet." from http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/ResultSet.html.

[Oracle, '12n] Oracle. (2012n). "Socket Communications." from http://www.oracle.com/technetwork/java/socket-140484.html.

[Oracle, '12o] Oracle. (2012o). "Transactions." from http://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html.

[Oracle, '12p] Oracle. (2012p). "UrlClassLoader." from http://docs.oracle.com/javase/1.4.2/docs/api/java/net/URLClassLoader.html.

[Oracle, '12q] Oracle. (2012q). "Uses of Reflection." from http://docs.oracle.com/javase/tutorial/reflect/index.html.

[Oracle, '12r] Oracle. (2012r). "Using Reflection." from http://java.sun.com/developer/technicalArticles/ALT/Reflection/.

[Pereira, '10] Pereira, Óscar Mortágua, Aguiar, Rui L. and Santos, Maribel Yasmina: "CRUD-DOM: A Model for Bridging the Gap Between the Object-Oriented and the Relational Paradigms."; ICSEA 2010 - Int. Conf. on Software Engineering and Applications, Nice, France (2010), 114-122.

[Pereira, '11a] Pereira, Óscar Mortágua, Aguiar, Rui L. and Santos, Maribel Yasmina: "An Adaptable Business Component Based on Pre-defined Business Interfaces."; 6th ENASE: Evaluation of Novel Approaches to Software Engineering, Beijing, China (2011a), 92-103.

[Pereira, '11b] Pereira, Óscar Mortágua, Aguiar, Rui L. and Santos, Maribel Yasmina: "CRUD-DOM: A Model for Bridging the Gap Between the Object-Oriented and the Relational Paradigms - an Enhanced Performance Assessment Based on a case Study." International Journal On Advances in Software, 4, (2011b), 158-180.

[Pereira, '12a] Pereira, Óscar Mortágua, Aguiar, Rui L. and Santos, Maribel Yasmina: "ACADA: Access Control-driven Architecture with Dynamic Adaptation." International Conference on Software Engineering and Knowledge Engineering, S. Francisco, CA, USA (2012a).

[Pereira, '12b] Pereira, Óscar Mortágua, Aguiar, Rui L. and Santos, Maribel Yasmina: "ORCA: Architecture for Business Tier Components Driven by Dynamic Adaptation and Based

on Call Level Interfaces."; 38th Euromicro Conf. on Software Engineering and Advanced Applications, Cesme, Izmir, Turkey (2012b), 183-191.

[Rizvi, '04] Rizvi, Shariq, Mendelzon, Alberto, Sudarshan, S. and Roy, Prasan: "Extending query rewriting techniques for fine-grained access control." Proceedings of the 2004 ACM SIGMOD international conference on Management of data, ACM, Paris, France (2004), 551-562.

[Samarati, '01] Samarati, Pierangela and Vimercati, Sabrina De Capitani di: "Access Control: Policies, Models, and Mechanisms." Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures, Springer-Verlag(2001), 137-196.

[Sandhu, '94] Sandhu, Ravi: "Database Access Controls." Auerbach Publishers, George Mason University, Fairfax : Center for Secure Information Systems and Department of Information and Software Systems Engineering (1994).

[Sandhu, '96] Sandhu, Ravi S., Coyne, Edward J., Feinstein, Hal L. and Youman, Charles E.: "Role-Based Access Control Models." Computer, 29, 2 (1996), 38-47.

[Sumathi, '07] Sumathi, S. and Esakkirajan, S.: "Fundamentals of Relational Database Management Systems"; Editora Springer, (2007).

[Vimercati, '08] Vimercati, S. DeCapitanidi, Foresti, S. and Samarati, P. (2008). Recent Advances in Access Control. Handbook of Database Security. M. Gertz and S. Jajodia, Springer US**:** 1-26.

[Zytrax, '12] Zytrax. (2012). "Regex." from http://www.zytrax.com/tech/web/regex.htm.